Optimización mediante Cúmulos de Partículas con Tamaño de Población Variable

Autor: María Victoria Leza

Directora: Prof. Lic. Laura Lanzarini

Tesina de Licenciatura en Sistemas
Facultad de Informática
UNIVERSIDAD NACIONAL DE LA PLATA

22 de mayo de 2008

Resumen

La resolución de problemas de optimización es de gran interés en la actualidad y ha motivado el desarrollo de diversos métodos informáticos para tratar de resolverlos.

La Optimización mediante Cúmulos de Partículas o PSO (Particle Swarm Optimization) es una metaheurística que ha sido utilizada exitosamente en la resolución de una amplia gama de problemas de optimización, incluyendo el entrenamiento de redes neuronales y la minimización de funciones. En su definición original, PSO utiliza, durante todo el proceso adaptativo, una población formada por un número fijo de soluciones.

El objetivo central de esta tesina es presentar una extensión original de PSO que incorpora los conceptos de edad y vecindad para permitir la variación del tamaño de la población. De esta forma, no es necesario definir a priori la cantidad de soluciones a utilizar, evitando así condicionar la calidad de la solución a obtener.

La variación del tamaño de la población se basa en una modificación del proceso adaptativo permitiendo el agregado y/o eliminación de individuos en función de su aptitud para resolver el problema planteado. Esto se realiza principalmente a través del concepto de edad que permite determinar el tiempo de permanencia de cada elemento dentro de la población. Además, dado que PSO tiende a poblar rápidamente las zonas exploradas con buen fitness, para no poblar excesivamente un mismo lugar del espacio de soluciones, se analiza el entorno de cada individuo y se eliminan las peores soluciones de las zonas muy pobladas.

El método aquí propuesto es aplicado a la resolución de algunas funciones complejas hallando mejores resultados que los que habitualmente se logran utilizando población de tamaño fijo.

Palabras Claves: Computación Evolutiva, Inteligencia de Cúmulo, Optimización de Cúmulo de Partículas, Optimización de Funciones.

Índice general

Capítulo 1

Introducción

La resolución de problemas de optimización es de gran interés en la actualidad y ha motivado el desarrollo de diversos métodos informáticos para tratar de resolverlos. La Optimización mediante Cúmulos de Partículas o PSO (Particle Swarm Optimization) es una metaheurística que ha sido utilizada exitosamente en la resolución de una amplia gama de problemas de optimización.

En este capítulo se presenta una introducción al concepto de Técnicas Metaeurísticas de Optimización, llegando a explotar la clasificación que las agrupa en aquellas basadas en la Trayectoria y aquellas basadas en la Población.

1.1. Técnicas Metaheurísticas de Optimización

La resolución de problemas de optimización, en el sentido de encontrar una mejor solución o al menos una solución aceptable, es un área de interés actual dada la aplicación en un sin número de situaciones de la vida real. Puede clasificarse un problema como "difícil", o NP-Completo, si [39]:

- El problema es de una naturaleza tal que no se conoce ningún método exacto para su resolución.
- Aunque existe un método exacto para resolver el problema, su uso es computacionalmente muy costoso o inviable.
- El método heurístico es más flexible que un método exacto, permitiendo, por ejemplo, la incorporación de condiciones de difícil modelización.

Debido a la gran importancia de este tipo de problemas, durante el transcurso de los años se han desarrollado varias técnicas para tratar de resolverlos. Una clasificación inicial es aquella que las agrupa en exactas y aproximadas o heurísticas (figura 1.1).

Las técnicas exactas garantizan encontrar la solución óptima para un problema en un tiempo limitado. El inconveniente de estos métodos es que el tiempo necesario para su

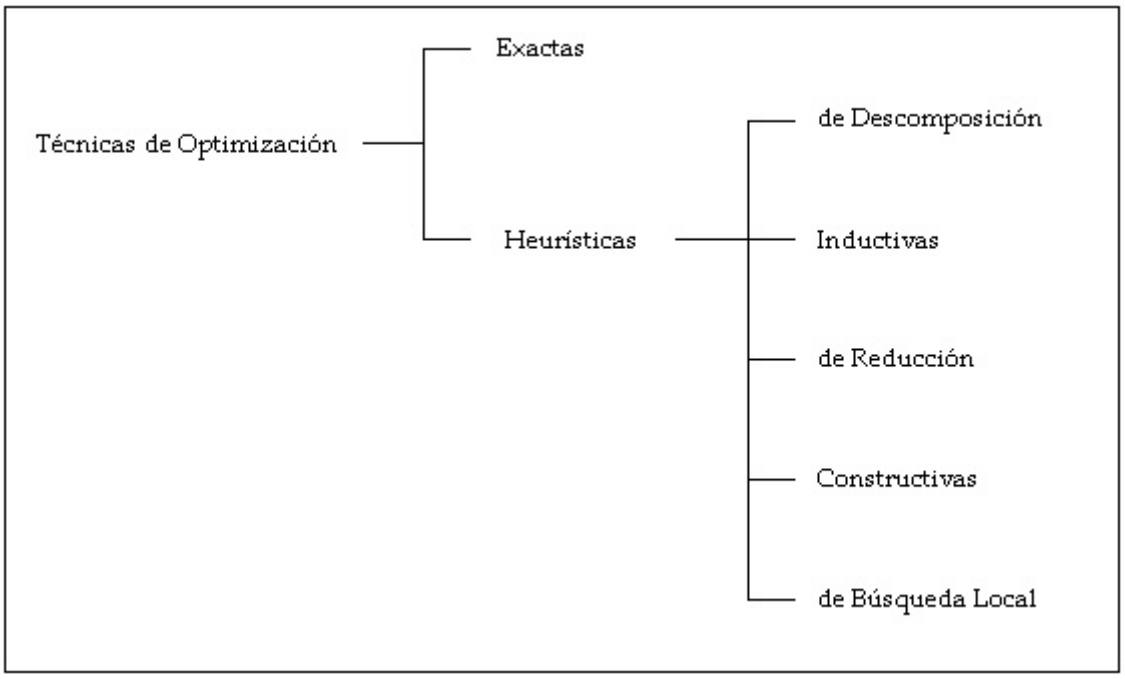


Figura 1.1: Clasificación de las Técnicas de Optimización

ejecución crece exponencialmente con relación al tamaño del problema, ocasionando que los problemas NP-Completos tengan tiempos de ejecución inalcanzables. Debido a estas limitaciones surgen los algoritmos aproximados o heurísticas.

La palabra heurística deriva del griego heuriskein, que significa encontrar o descubrir, aunque el término fue definido más precisamente por Reeves [47]:

"Una heurística es una técnica que busca soluciones buenas (es decir, casi óptimas) a un costo computacional razonable, aunque sin garantizar factibilidad u optimalidad de las mismas. En algunos casos, ni siquiera puede determinar qué tan cerca del óptimo se encuentra una solución factible en particular."

Existen muchos métodos heurísticos de diferentes naturalezas, y muchos de ellos han sido diseñados para un problema específico sin posibilidad de generalización, por lo que es complicado dar una clasificación completa. El siguiente esquema trata de brindar una clasificación en dónde poder ubicar a los heurísticos más conocidos [39]:

- Métodos de Descomposición: La idea principal de estos métodos es la descomposición del problema original en subproblemas de solución más sencilla.

- Métodos Inductivos: La idea de estos métodos es generalizar de versiones pequeñas o más sencillas de resolver al problema completo.

- Métodos de Reducción: Estos métodos identifican las propiedades que se cumplen mayoritariamente por las buenas soluciones al problema y las introducen como restricciones del mismo. Esto tiene por objetivo poder reducir el espacio de soluciones simplificando el problema, aunque tiene como riesgo la posibilidad de dejar afuera soluciones óptimas del problema original.

- Métodos Constructivos: Consisten en construir literalmente paso a paso una solución del problema. Usualmente son métodos deterministas y suelen estar basados en la mejor elección en cada iteración.

- Métodos de Búsqueda Local: A diferencia de los métodos anteriores, estos métodos comienzan con una solución del problema y la mejoran progresivamente. El procedimiento realiza en cada paso un movimiento de una solución a otra con mejor valor. El método finaliza cuando, para una solución, no existe ninguna solución accesible que la mejore.

Como otra clasificación, surgida en base a los métodos constructivos y los de búsqueda local, se puede hablar de los procedimientos metaheurísticos, cuya idea básica es combinar diferentes métodos heurísticos a un nivel más alto para conseguir una exploración del espacio de búsqueda de forma eficiente y efectiva. Este término fue introducido por primera vez en [22] por Glover.

Por lo tanto, pueden caracterizarse a las metaheurísticas como [1, 6, 23]:

- Estrategias o plantillas de alto que establecen el proceso de búsqueda general y tienen por objetivo encontrar soluciones casi óptimas.

- Algoritmos aproximados y no determinísticos, que pueden incorporar mecanismos para evitar óptimos locales.

- Soluciones no específicas al problema que se intenta resolver, incorporando el conocimiento específico del problema o la experiencia para guiar la búsqueda. Utilizan funciones de bondad o fitness para cuantificar el grado de adecuación de una determinada solución.

- Soluciones que poseen un amplio espectro de aplicación, desde técnicas sencillas, por ejemplo una búsqueda local, a técnicas complejas, por ejemplo procesos de aprendizaje.

Existen diversas clasificaciones para agrupar a las metaheurísticas, pero en lo que continúa del capítulo se detallará la agrupación que divide a las metaheurísticas en aquellas basadas en la trayectoria y aquellas basadas en la población, como se muestra en la figura 1.2.

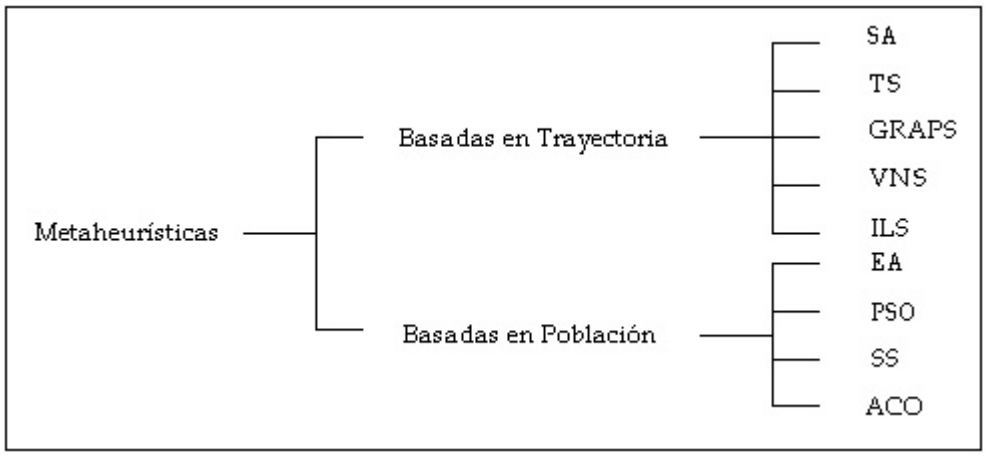


Figura 1.2: Clasificación de Metaheurísticas

1.1.1. Metaheurísticas basadas en la Trayectoria

La idea principal de este tipo de metaheurísticas es que parte de una solución inicial y, mediante la exploración de su entorno, actualizan la solución actual formando una trayectoria en este espacio de soluciones. Estas técnicas surgen como extensiones de los métodos de búsqueda local simples a los que se les añade alguna característica para escapar de los mínimos locales [45].

- La Búsqueda Tabú o Tabu Search (TS) es una metaheurística cuyos fundamentos fueron introducidos en [22]. La idea básica es el uso explícito de un historial de la búsqueda o memoria que le permite escapar de los óptimos locales y evitar exploraciones excesivas sobre una misma región. Esta memoria es implementada como una lista tabú, normalmente con una metodología FIFO, donde se mantienen las soluciones visitadas más recientemente para excluirlas de los próximos movimientos. Por lo tanto, en cada iteración se elige la mejor solución entre las permitidas y se la añade a la lista tabú.
  Como desventaja de esta técnica, se puede observar que el proceso permite la pérdida de información, ya que buenas soluciones pueden ser excluídas del conjunto permitido. Para reducir esto, se puede definir un criterio que permita a una solución estar dentro del conjunto de soluciones permitidas aún cuando figure en la lista tabú.

- La Búsqueda en Vecindario Variable o Variable Neighborhood Search (VNS) es una metaheurística propuesta en [43], en la cual propone dentificar, para un conjunto definido al incio del algoritmo, diferentes vecindarios con diferentes proximidades (usualmente crecientes). La idea básica es explorar el vecindario actual hasta que no sea posible mejorar la solución encontrada, momento en el que se decide cambiar de vecindario en busca de soluciones aún más óptimas.

- El Enfriamiento Simulado o Simulated Annealing (SA), inicialmente presentado en [36], tiene como idea principal simular el proceso físico del calentamiento de metales y

del cristal. Se propone una similitud entre una buena estructura cristalina de metales y una buena estructura de soluciones para problemas de optimización combinatoria. Este algoritmo tiene por objetivo minimizar una función que representa la energía del sistema. Para evitar quedar atrapado en un óptimo local, el algoritmo permite elegir una solución peor que la solución actual.
En cada iteración se elige, a partir de la solución actual s, una solución s_0 del vecindario $N(s)$. Si s_0 es mejor que s, se sustituye s por s_0 como solución actual. Si la solución s_0 es peor, entonces es aceptada con una determinada probabilidad que depende de la temperatura actual T y de la variación en la función de fitness, $f(s_0) - f(s)$.
Al comienzo, la temperatura es alta, cualquier transición entre estados es permitida y soluciones que empeoren la función objetivo pueden ser aceptadas con mayor probabilidad que más tarde cuando la temperatura haya disminuido. La temperatura del sistema es controlada de acuerdo al enfriamiento sucesivo (función logarítmica) y recalentamientos periódicos, que permiten escapar de óptimos locales (estructura de soluciones).

- El Procedimiento de Búsqueda Miope Aleatorizado y Adaptativo o The Greedy Randomized Adaptive Search Procedure (GRASP) es una metaheurística que combina heurísticos constructivos y de búsqueda local [18]. En su versión clásica cada iteración comprende dos fases: una fase constructiva que tiene como resultado una solución viable, aunque no necesariamente un óptimo local, y una búsqueda local, durante la cual se examinan vecindades de esta solución. La iteración termina al momento de llegar a un óptimo local. El algoritmo continúa, guardando la mejor solución encontrada para cada iteración, y termina cuando se alcanza algún criterio prefijado.

- En la Búsqueda Local Iterada o Iterated Local Search (ILS) propuesta en [52] se parte de una solución inical, donde una alteración sobre la misma genera una solución intemedia desde dónde se aplica un método de búsqueda local para mejorarla. Luego de esta búsqueda, el óptimo local encontrado durante el proceso puede ser aceptado como nueva solución actual si pasa un test de aceptación.

1.1.2. Metaheurísticas basadas en la Población

Los métodos basados en población trabajan con un conjunto de soluciones (población) en cada iteración, teniendo de esta manera un enfoque diferente que el de los métodos vistos anteriormente que únicamente utilizan un punto del espacio de búsqueda por iteración. El resultado final de este tipo de algoritmos depende fuertemente de la forma en que manipula la población [45].

- Los Algoritmos Evolutivos o Evolutionary Algorithms (EA) [4] engloban una serie de técnicas inspiradas biológicamente en la teoría Neo-Darwiniana de la Evolución. Esta familia de técnicas siguen un proceso en el cual una población de individuos (inicialmente aleatoria) es evolucionada utilizando mecanismos de selección, reproducción y mutación de individuos, dónde un individuo es una posible solución al

problema. Cada individuo en la población tiene asignado, por medio de una función de aptitud (fitness), una medida que cuantifica cuan factible es la solución para el problema que se intenta resolver, dónde este valor es el pivot que utiliza el algoitmo para guiar la búsqueda. Estos algoritmos establecen un equilibrio entre la explotación de buenas soluciones (fase de selección) y la exploración de nuevas zonas del espacio de búsqueda (fase de reproducción).

- La Búsqueda Dispersa o Scatter Search (SS) [23] es una metaheurística presentada en [21]. El algoritmo se basa en mantener un conjunto de soluciones posibles pequeño, llamado "conjunto de referencia", que se caracteriza tanto por contener buenas soluciones como soluciones con menos aptitud, brindando así diversidad a la búsqueda. Durante el algoritmo, este conjunto se divide en subconjuntos de soluciones a las cuales se les aplica una operación de recombinación y mejora (mecanismos de búsqueda local).

- Los Algoritmos basados en Colonias de Hormigas o Ant Colony Optimization (ACO) [15] constituyen una metaheurística que simulan el comportamiento de las colonias de hormigas durante el proceso de búsqueda de alimentos. Durante este proceso, la colonia entera trata de minimizar la distancia que existe entre la fuente de alimentos y el nido dónde lo consumen. Si bien cada hormiga tiene capacidades básicas, la colonia en conjunto logra el comportamiento inteligente.
  Inicialmente, las hormigas exploran de manera semi-aleatoria el área cercana a su nido. Tan pronto como una hormiga encuentra la comida, la lleva al nido depositando, tanto en el camino de ida como en el de vuelta, una sustancia química denominada feromona. Esta sustancia ayudará al resto de las hormigas a encontrar la fuente de alimento más cercana al nido, ya que, cada hormiga, tiene biológicamente una tendencia a seguir un camino cuanto mayor sea la cantidad de feromonas depositadas en el mismo. Este comportamiento de grupo es el que se trata de simular el algoritmo, imitando el rastro de feromona mediante un modelo probabilístico.

- Los Algoritmos basados en Cúmulos de Partículas o Particle Swarm Optimization (PSO) [35] es una metaheurística inspirada en el comportamiento social del vuelo de las bandadas de aves o el desplazamiento de los cardúmenes de peces. La idea principal del algoritmo es simular la influencia social del entorno que posee cada agente durante la selección de la próxima dirección a seguir para alcanzar una nueva posición dentro del espacio de soluciones. Esta selección no sólo es influenciada por el conocimiento de su entorno, sino también por las experiencias anteriores del agente.

Capítulo 2

Algoritmos basados en Cúmulos de Partículas

La Optimización mediante Cúmulos de Partículas o PSO (Particle Swarm Otimization) es una metaheurística que ha sido utilizada exitosamente en la resolución de una amplia gama de problemas de optimización, incluyendo el entrenamiento de redes neuronales y la minimización de funciones.

En este capítulo se realiza una descripción más detallada de este tipo de algoritmos y se presentan sus principales características y aspectos avanzados, tomando como primera fuente de consulta a la tesis de García Nieto [45].

2.1. Introducción

En su concepción, por Kennedy y Eberhart en [33], la Optimización mediante Cúmulos de Partículas simula el comportamiento de las aves volando al azar en búsqueda de alimentos en una zona. Si en su vuelo sólo registraron una porción de alimentos, y todos los pájaros no saben si realmente es alimento, entonces ¿cuál es la mejor estrategia para encontrarlo? Lo más efectivo es seguir el pájaro que está más cerca de el (figura 2.1).

Un "Algoritmo basado en Optimización de Cúmulos de Partículas" o Particle Swarm Optimization es una técnica metaheurística basada en la idea de simular el comportamiento social de una población de individuos, donde cada individuo intenta mejorar sus conocimientos con interacciones dentro de su entorno social. Este algoritmo se basa en la idea expuesta en [35] de que los individuos que conviven en una sociedad tienen una opinión que es parte de un conjunto de creencias compartido (el espacio de búsqueda) por todos los posibles individuos. Cada individuo puede modificar su opinión dependiendo de:

- Su conocimiento sobre el entorno (su valor de fitness).
- Su conocimiento histórico o experiencias anteriores (su memoria o conocimiento cognitivo).

Figura 2.1: Ejemplos de Inteligencia Swarm en la naturaleza

- El conocimiento histórico o experiencias anteriores de los individuos situados en su vecindario (su conocimiento social).

En base a ésto, cada individuo adapta su conjunto de creencias según las creencias de aquellos con más éxito de su entorno, originando así una cultura en dónde sus individuos tienen un conjunto de creencias estrechamente relacionado.

El cúmulo de partículas (swarm) se puede ver como un sistema multiagente, dónde las partículas son agentes simples que se mueven por el espacio de búsqueda, guardando y comunicando la mejor solución que han encontrado. Cada partícula tiene un fitness, una posición y un vector velocidad que dirige su "movimiento". El movimiento de las partículas por el espacio está guiado por las partículas óptimas en el momento actual.

2.2. Algoritmos basados en Optimización de Cúmulos de Partículas (PSO)

Un algoritmo PSO consiste en un proceso iterativo y estocástico que opera sobre un cúmulo de partículas. La posición de cada partícula representa una solución potencial al problema que se está resolviendo.

Cada partícula p_i está compuesta por tres vectores y dos valores de fitness:

- El vector $x_i = (x_{i1}, x_{i2}, \ldots, x_{in})$ almacena la posición actual de la partícula en el espacio de búsqueda.
- El vector $pBest_i = (p_{i1}, p_{i2}, \ldots, p_{in})$ almacena la posición de la mejor solución encontrada por la partícula hasta el momento.
- El vector velocidad $v_i = (v_{i1}, v_{i2}, \ldots, v_{in})$ almacena el gradiente (dirección) según el cual se moverá la partícula.
- El valor de fitness $fitness\_x_i$ almacena el valor de aptitud de la solución actual (vector x_i).

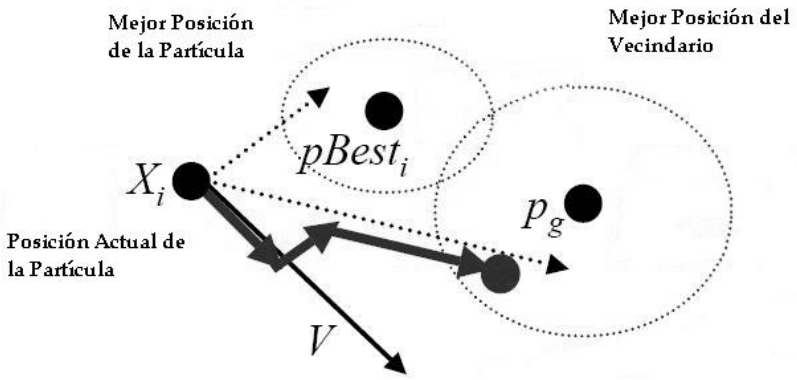


Figura 2.2: Movimiento de una partícula en el espacio de soluciones

- El valor de fitness $fitness\_pBest_i$ almacena el valor de aptitud de la mejor solución local encontrada hasta el momento (vector $pBest_i$).

La posición de una partícula se actualiza se la siguiente forma:

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{2.1}$$

La figura 2.2 muestra gráficamente el movimiento de la partícula.

Como se explicó anteriormente, el vector velocidad se modifica teniendo en cuenta su experiencia y la de su entorno. La expresión es la siguiente:

$$v_i(t+1) = w.v_i(t) + \varphi_1.rand_1.(pBest_i - x_i(t)) + \varphi_2.rand_2.(g_i - x_i(t)) \tag{2.2}$$

donde w representa el factor de inercia [51], φ_1 y φ_2 son constantes de aceleración que representan la importancia que se le dará al conocimiento cognitivo y social respectivamente, $rand_1$ y $rand_2$ son valores aleatorios pertenecientes al intervalo (0,1) y g_i representa la posición de la partícula con el mejor $pBest\_fitness$ del entorno de p_i (lBest o localbest) o de todo el cúmulo (gBest o globalbest). Los valores de w, φ_1 y φ_2 son importantes para asegurar la convergencia del algoritmo. Para más detalles sobre la elección de estos valores puede consultar [5, 10].

La figura 2.3 contiene el algoritmo PSO básico.

2.2.1. Tipos de Algoritmos basados en PSO

En base a diferentes configuraciones de los parámetros del algoritmo, se pueden originar diferentes variantes de PSO. Estas versiones, agrupadas según las configuraciones que les dan origen, son:

Pop= $CrearPoblacion(N)$;
while (no se alcance la condición de terminación) do
    for i = 1 to size(Pop) do
        Evaluar Partícula x_i del Cúmulo Pop;
        if fitness(x_i) es mejor que el $fitness(pBest_i)$ then
            $pBest_i \leftarrow x_i$; $fitness(pBest_i) \leftarrow fitness(x_i)$;
        end if
    end for
    for i = 1 to size(Pop) do
        Elegir g_i de acuerdo al criterio de vecindario usado;
        $v_i \leftarrow w.v_i + (\varphi_1.rand_1.(pBest_i - x_i) + \varphi_2.rand_2.(g_i - x_i)$;
        $x_i \leftarrow x_i + v_i$;
    end for
end while
Salida : la mejor solución encontrada;

Figura 2.3: Algoritmo PSO básico

1. En base a la influencia de los pesos cognitivo y social (φ_1 y φ_2 respectivamente) en la dirección que toma la velocidad de una partícula en movimiento (ecuación 2.1), se pueden generar cuatro tipos de variantes para PSO:

    - Modelo Completo: φ_1; $\varphi_2 > 0$. Tanto la influencia cognitiva como la social intervienen en el movimiento.
    - Modelo sólo Cognitivo: $\varphi_1 > 0$ y $\varphi_2 = 0$. Únicamente la influencia cognitiva interviene en el movimiento.
    - Modelo sólo Social: $\varphi_1 = 0$ y $\varphi_2 > 0$. Únicamente la influencia social interviene en el movimiento.
    - Modelo sólo Social exclusivo: $\varphi_1 = 0$, $\varphi_2 > 0$ y $g_i \neq x_i$. La posición actual de la partícula no puede ser la mejor de su entorno.

2. En base al vecindario, se pueden generar dos variantes más del algoritmo:

    - *PSOLocal* : En esta variante el conocimiento social de la partícula es evaluado solo en el vecindario de la misma.
    - *PSOGlobal* : En esta, en cambio, el conocimiento social para cada partícula es evaluado en la población completa.

    En estos casos, el tamaño de la vecindad influye en la velocidad de convergencia del algoritmo así como en la diversidad de los individuos de la población, ocasionando que a mayor tamaño de vecindad, la convergencia sea más rápida y la diversidad de la población menor.

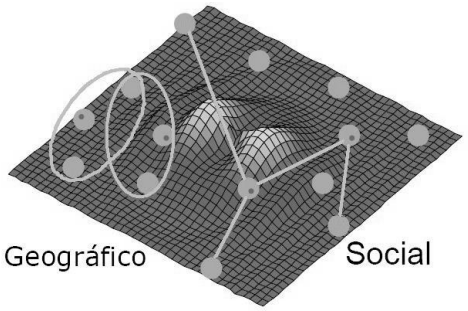


Figura 2.4: Ejemplos de entornos sociales y geográficos en un espacio de soluciones

2.2.2. Topologías de Cúmulos de Partículas

Otra cuestión importante del algoritmo es la manera en la que una partícula interacciona con las demás partículas de su vecindario. Las topologías definen el entorno de interacción de una partícula con su vecindario, originando dos tipos de entornos:

- Geográficos: donde el entorno del cada partícula está formado por aquellos vecinos que se encuentran más cerca de su posición actual.
- Sociales: donde el entorno de cada partícula es definido sin importar su posición en el espacio.

En la figura 2.4 se muestran algunos ejemplos de estos entornos.

2.3. Aspectos avanzados de los Algoritmos basados en PSO

Como principal problema de los algoritmos basados en Cúmulos de Partículas puede verse que si no se fija una magnitud correcta para v_{max}, la velocidad puede crecer demasiado y ocasionar que las partículas tengan movimientos excesivos en el espacio de búsqueda. Como se detalla en [16], existen dos métodos para controlar el excesivo crecimiento de las velocidades: un factor de inercia ajustado dinámicamente y un coeficiente de constricción.

La idea de tener un factor de inercia w que influya en la velocidad dinámicamente indica que la inercia se reduce gradualmente a lo largo del tiempo (iteraciones del algoritmo) [41], como muestra la siguiente ecuación 2.3:

$$w = w_{max} - \frac{w_{max} - w_{min}}{iter_{max}}.iter \tag{2.3}$$

donde w_{max} es el peso inicial y w_{min} el peso final, $iter_{max}$ es el número máximo de iteraciones e $iter$ es la iteración actual. w debe mantenerse entre 0.9 y 1.2, ya que valores altos provocan una mayor diversificación y valores bajos una mayor intensificación.

Por otro lado, el coeficiente de constricción introduce una nueva ecuación para la actualización de la velocidad (ecuaciones 2.4 y 2.5) asegurando la convergencia [16].

$$v_i(t+1) = K[w.v_i(t) + \varphi_1.rand_1.(pBest_i - x_i(t)) + \varphi_2.rand_2.(g_i - x_i(t))] \tag{2.4}$$

$$K = \frac{2}{2 - \varphi - \sqrt{\varphi^2 - 4\varphi}} \tag{2.5}$$

Es sabido que, otro aspecto a considerar es el tamaño del cúmulo, si el tamaño de la población es pequeño, el algoritmo puede converger demasiado rápido, pero si el tamaño es muy grande, la performance del algoritmo suele verse degradada. Si bien hoy existen algunas heurísticas para adaptar dinámicamente el tamaño del cúmulo, ninguna tiene un funcionamiento totalmente óptimo. En los próximos capítulos se presentarán ciertas modificaciones al algoritmo PSO original para poder mitigar este problema de manera más eficiente.

Finalmente, existen trabajos en dónde se utilizan valores adaptativos para los coeficientes de aprendizaje cognitivo y social, (φ_1 y φ_2), dónde se definen en base a la calidad de la propia partícula y del entorno.

2.4. PSO según la Codificación

El algoritmo PSO de la figura 2.3 itera sobre el cúmulo modificando la posición actual de cada partícula a través del operador de movimiento. Este pseudocódigo define el algoritmo PSO básico, pero dependiendo de la implementación que tengan las partículas, es decir, la representación de las soluciones, y las fases de actualización de velocidad y movimiento, se obtendrá una de las tres variantes principales del PSO: codificación continua, binaria y permutaciones de enteros.

2.4.1. PSO para Codificación Continua

Esta es la versión expuesta en la sección 2.2, y es también la más conicida del algoritmo, por lo que en esta sección, sólo se realizan algunos comentarios más específicos de la codificación continua.

Para no confundir el ámbito de aplicación de los operadores sociales, se representa a la mejor partícula del vecindario como g, pudiendo ser ésta bien $gBest$ o bien $lBest$ dependiendo de la versión elegida.

Representación de las Partículas

Cada partícula, para cualquier versión del PSO, está formada como mínimo por:

$$p = \{x, pBest, v\} \tag{2.6}$$

donde

- x es un vector real que representa un punto (posición) en el espacio de búsqueda, en la mayoría de los casos, una solución.

- $pBest$ es un vector que tiene la misma codificación que x pues representa la mejor posición (solución) encontrada por la partícula p hasta el momento.

- v es un vector que representa la velocidad de la partícula, es decir, la dirección de la partícula o el vector gradiente en el espacio de búsqueda. En cada iteración, la velocidad es actualizada mediante la suma con los factores cognitivo y social, por lo que la magnitud de la velocidad puede crecer de forma importante durante la ejecución y las partículas se moverían muy rápido por el espacio. Por este motivo, se suele acotar el valor de la velocidad en un rango $[v_{max}, -v_{max}]$, siendo v_{max} la velocidad máxima inicial de cada componente del vector velocidad.

Operador Actualización de Velocidad

La actualización de la velocidad de cada partícula para esta codificación de PSO está dada por la ecuación:

$$v = w.v + \varphi_1.rand_1.(pBest - x)) + \varphi_2.rand_2.(g - x) \tag{2.7}$$

donde

- Los vectores x, $pBest$, g y v son reales, las operaciones de suma, resta y multiplicación se realizan sin ningún tipo de transformación.

- Los valores aleatorios obtenidos en $rand_1$ y $rand_2$ deben estar comprendidos en $[0, 1]$.

Operador Movimiento

El operador movimiento de la partícula es la suma del vector posición x y el vector velocidad v, generando la nueva posición en x, lugar de partida de la partícula en la siguiente iteración y posible solución al problema que se busca resolver.

$$x = x + v \tag{2.8}$$

2.4.2. PSO para Codificación Binaria

Existen una gran cantidad de problemas de optimización que requieren de una representación binaria de sus soluciones. Por esto, Kennedy y Eberhart propusieron en [34] una adaptación inicial del PSO binario donde la posición de la partícula es un vector cadena de bits, mientras que la velocidad se representa mediante un vector real.

El operador movimiento, también depende de la velocidad, aunque aquí se define de la siguiente manera: si la velocidad es alta respecto a un valor umbral, el nuevo valor de la posición x será 1, y si es baja será 0 (véase la ecuación 2.11). El valor umbral (representado por ρ) está comprendido en el intervalo $[0, 1]$. Como el umbral está en ese rango, también debe normalizarse el valor de la velocidad al mismo rango, para ello se suele utilizar la función sigmoidal (ecuación 2.9).

$$sig(v^k) = \frac{1}{1 + exp(-v^k)} \tag{2.9}$$

De esta manera, en cada iteración k (siendo k la longitud de x) se obtienen los valores 0 o 1 que corresponden a cada posición de la partícula. El algoritmo PSO binario se describe con las siguientes ecuaciones 2.10 y 2.11 de actualización de velocidad y movimiento:

$$v^k = w.v^k + \varphi_1.rand_1.(pBest - x^k)) + \varphi_2.rand_2.(g - x^k) \tag{2.10}$$

$$\rho^k < sig(v^k)\ entonces\ x^k = 1;\ en\ otro\ caso\ x^k = 0 \tag{2.11}$$

donde, ρ^k es el vector de valores umbrales comprendidos en $[0, 1]$.

Como se puede observar, la transformación con la función sigmoidal lleva toda la información sobre la dirección de la partícula a dos únicos niveles de decisión limitados por el valor umbral, ocasionando pérdida de información. En vista de esto, Clerc en [9] consigió mitigar un poco este problema proponiendo una versión más eficiente que llamó *Derivation* 0 (figura 2.5), donde la trasformación se aplica mediante operadores de aritmética modular.

Representación de las Partículas

La posición de una partícula está representada por una cadena de ceros y unos. Esta cadena es originada gracias a la suma del vector velocidad, cuyos valores son 0, 1 mod_2, -1 mod_2. Al utilizar velocidades que tengan únicamente estos tres posibles valores $\{-1, 0, 1\}$ se combinan esas tres tendencias para que el resultado sea siempre 0 o 1 utilizando la función módulo.

Operador Actualización de Velocidad

Para actualizar el vector velocidad se utiliza, además de las ecuaciones tradicionales, la ecuación 2.13, que se encarga de realizar la transformación modular de los valores de velocidad. Esta operación se realiza tras el movimiento de la partícula, ya que en el movimiento interviene la velocidad sin transformación.

$$v = w.v + \varphi_1.rand_1.(pBest - x)) + \varphi_2.rand_2.(g - x) \tag{2.12}$$

$$v = (3 + v) mod_3 - 1 \tag{2.13}$$

Los coeficientes φ_1 y φ_2 son obtenidos según una distribución uniforme en el rango $[-1, 1]$. Mediante la operación mod_3 se transforma a los nuevos valores de velocidad v en valores del conjunto $\{-1, 0, 1\}$.

Pop= $CrearPoblacion(N)$;
while (no se alcance la condición de terminación) do
    for i = 1 to size(Pop) do
        Evaluar Partícula x_i del Cúmulo Pop;
        if fitness(x_i) es mejor que el $fitness(pBest_i)$ then
            $pBest_i \leftarrow x_i$; $fitness(pBest_i) \leftarrow fitness(x_i)$;
        end if
    end for
    for i = 1 to size(Pop) do
        Elegir g_i de acuerdo al criterio de vecindario usado;
        $v_i \leftarrow w.v_i + (\varphi_1.rand_1.(pBest_i - x_i) + \varphi_2.rand_2.(g_i - x_i)$;
        $x_i \leftarrow x_i + v_i$;
        $x_i \leftarrow (4 + x_i) mod_2$;
        $v_i \leftarrow (3 + v_i) mod_3 - 1$;
    end for
end while
Salida : la mejor solución encontrada;

Figura 2.5: Algoritmo PSO con Codificación Binaria Derivation 0

Operador Movimiento

De la misma manera que con la actualización de la velocidad, la operación de movimiento sobre la posición actual luego necesita una trasformación modular sobre los valores de la nueva posición.

$$x = x + v \tag{2.14}$$

$$x = (4 + x) mod_2 \tag{2.15}$$

Al aplicar mod_2 a los valores de la nueva posición x de la partícula, se transforman modularmente a ceros y unos, y con la suma del valor cuatro a la posición x, se garantizan únicamente valores (ceros o unos) positivos, ya que despues de la suma de la velocidad se pueden generar valores negativos.

2.4.3. PSO para Permutaciones de Enteros

Los problemas basados en permutaciones de enteros son otro gran subconjunto dentro de los problemas de optimización, por esto Clerc en [9] desarrolló una novedosa versión del PSO para este tipo de problemas que se datellará a continuación (figura 2.6). Como característica principal, en esta versión se trabaja con un nuevo conjunto de operadores, los cuales son utilizados en las ecuaciones de actualización de velocidad y de movimiento: la suma $(\oplus, \circ)$, la diferencia $(\ominus)$ y la multiplicación $(\otimes\neg)$ de permutaciones.

Pop= $CrearPoblacion(N)$;
while (no se alcance la condición de terminación) do
    for i = 1 to size(Pop) do
        Evaluar Partícula x_i del Cúmulo Pop;
        if fitness(x_i) es mejor que el $fitness(pBest_i)$ then
            $pBest_i \leftarrow x_i$; $fitness(pBest_i) \leftarrow fitness(x_i)$;
        end if
    end for
    for i = 1 to size(Pop) do
        Elegir g_i de acuerdo al criterio de vecindario usado;
        $v_i \leftarrow v_i \circ \varphi_1 \otimes (pBest_i \ominus x_i) \circ \varphi_2 \otimes (g_i \ominus x_i)$;
        $x_i \leftarrow x_i \oplus v_i$;
    end for
end while
Salida : la mejor solución encontrada;

Figura 2.6: Algoritmo PSO para Permutaciones de Enteros

Representación de las Partículas

Cada posición x de un partícula está formada por una lista de n posibles valores enteros sin repeticiones ni omisiones, donde n es también la longitud de x. Un sencillo ejemplo de longitud 8 puede ser:

$$x = (8, 4, 1, 5, 3, 2, 6, 7)$$

La velocidad v es una lista de pares de enteros $(i \rightarrow j)$, donde cada uno representa un intercambio a realizar sobre los elementos de x. Por ejemplo, si se aplica el intercambio del par $(3 \rightarrow 1)$ a la posición del ejemplo anterior, se obtiene la nueva posición $x = (8, 4, 3, 5, 1, 2, 6, 7)$. Por lo tanto, el movimiento de una partícula se consigue aplicando sucesivamente los pares de la lista velocidad sobre la lista posición. Por ejemplo, si se tiene la siguiente lista velocidad:

$$v = [(2 \rightarrow 1), (5 \rightarrow 5), (1 \rightarrow 4), (6 \rightarrow 8)]$$

y se aplica a la posición anterior, se obtiene:

1. $x' = (8, 4, 2, 5, 3, 1, 6, 7)$ al aplicar $(2 \rightarrow 1)$.
2. El par $(5 \rightarrow 5)$ no realiza ninguna permutación sobre la posición.
3. $x'' = (8, 1, 2, 5, 3, 4, 6, 7)$ al aplicar sobre x' el par $(1 \rightarrow 4)$.
4. $x''' = (6, 1, 2, 5, 3, 4, 8, 7)$ al aplicar sobre x' el par $(6 \rightarrow 8)$.

Finalmente, se puede decir que el tamaño de la velocidad $|v|$ es el número de pares que realizan intercambios, excluyendo los que no hacen nada, es decir, los pares $(i \rightarrow i)$. Para el ejemplo anterior $|v| = 3$.

Operador Actualización de Velocidad

La ecuación 2.16 muestra la actualización de velocidad para esta codificación. Esta ecuación tiene el mismo formato que para las versiones antes vistas, aunque los operadores de suma, diferencia y multiplicación para permutaciones de enteros son diferentes.

$$v_i \leftarrow v_i \circ \varphi_1 \otimes (pBest_i \ominus x_i) \circ \varphi_2 \otimes (g_i \ominus x_i) \tag{2.16}$$

Operador Resta de Posiciones ($\ominus$)

La resta de dos posiciones da como resultado una lista velocidad. Un par i está formado por el $i-$ésimo valor del segundo operando y el $i-$ésimo valor de primer operando de la resta. Por ejemplo, si se restan las siguientes posiciones:

$$(8, 4, 2, 5, 3, 1, 6, 7) \ominus (6, 1, 2, 5, 3, 4, 8, 7)$$

el resultado será la velocidad:

$$[(6 \rightarrow 8), (1 \rightarrow 4), (2 \rightarrow 2), (5 \rightarrow 5), (3 \rightarrow 3), (4 \rightarrow 1), (8 \rightarrow 6), (7 \rightarrow 7)]$$

Operador Suma de Velocidades ($\circ$)

La suma de dos velocidades consiste en aplicar la relación transitiva sobre los pares de las listas de dichas velocidades, para obtener, nuevamente, otra lista velocidad. Para que se produzca un nuevo par, deben coincidir los valores final e inicial de los pares involucrados: los pares $(i \rightarrow j)$ y $(j \rightarrow k)$ se relacionan formando el par $(i \rightarrow k)$. De lo contrario, si dos pares correspondientes no se pueden relacionar, el resultado será el primer par. A continuación se muestra un ejemplo:

$$[(2 \rightarrow 1), (5 \rightarrow 5), (1 \rightarrow 4), (6 \rightarrow 8)] \circ [(1 \rightarrow 4), (5 \rightarrow 5), (4 \rightarrow 2), (7 \rightarrow 5)]$$

dando como resultado

$$[(2 \rightarrow 4), (5 \rightarrow 5), (1 \rightarrow 2), (6 \rightarrow 8)]$$

Operador Producto Coeficiente Velocidad ($\otimes$)

El producto coeficiente velocidad es la multiplicación de un coeficiente real φ y una lista velocidad, que se realiza de la siguiente manera:

- Si $\varphi \epsilon [0, 1]$, se obtiene $\varphi' = rand(0, 1)$ para $\begin{cases} \varphi' \leq \varphi & \Rightarrow \quad (i \rightarrow j) \rightarrow (i \rightarrow i) \\ \varphi' > \varphi & \Rightarrow \quad (i \rightarrow j) \rightarrow (i \rightarrow j) \end{cases}$
- Si $\varphi > 1$, tal que $\varphi = k + \varphi'$, con k entero y $\varphi' < 1$, se realiza velocidad más velocidad k veces y coeficiente φ' por velocidad una vez.

Por ejemplo, si multiplicamos:

$$0,5 \otimes \neg[(2 \rightarrow 4), (5 \rightarrow 5), (4 \rightarrow 2), (6 \rightarrow 8)]$$

para la secuencia de valores aleatorios (φ') 0.6, 0.2, 0.9 y 0.1 se obtiene como resultado:

$$[(2 \rightarrow 4), (5 \rightarrow 5), (4 \rightarrow 2), (6 \rightarrow 6)]$$

Operador Movimiento

Al igual que en las versiones anteriores del PSO, el movimiento es la suma de la velocidad a la posición de la partícula (ecuación 2.17).

$$x_i \leftarrow x_i \oplus v_i \tag{2.17}$$

Operador Suma de Posición con Velocidad ($\oplus$)

Con este operador, se permutan sucesivamente los valores de la posición de una partícula para generar una nueva posición. Esto se realiza en base a cada par ($i \leftarrow j$) de la lista de velocidad, que indica un intercambio de los elementos i y j, dando como resultado la nueva posición a la que se mueve la partícula. Por ejemplo, aplicando esta operación a:

$$x = (8, 4, 1, 5, 3, 2, 6, 7) \oplus v = [(2 \rightarrow 4), (5 \rightarrow 5), (1 \rightarrow 2), (6 \rightarrow 8)]$$

se obtiene como resultado

$$x = (6, 1, 2, 5, 3, 4, 8, 7)$$

Capítulo 3

Algoritmos Genéticos

En los años 1960, gracias a John Holland, surgió una de las líneas más prometedoras de la Inteligencia Artificial: los Algoritmos Genéticos, llamados así porque se inspiran en la evolución biológica y su base genético-molecular. Los algoritmos genéticos forman parte de una familia denominada Algoritmos Evolutivos, que incluye las Estrategias de Evolución, la Programación Evolutiva y la Programación Genética.

En este capítulo se presenta una descripción profunda de esta clase de algoritmos, llegando a analizar las variantes de representación del problema y los operadores que permiten la exploración del espacio de soluciones.

3.1. Introducción

La naturaleza genera seres perfectamente adaptados a su entorno. En función de éste, los seres se han adaptado a lo largo de miles de generaciones perfeccionandose gracias a pequeños saltos y los individuos resultantes se extinguieron compitiendo al lado de otros más aptos para sobrevivir. "La selección natural obra solamente mediante la conservación y acumulación de pequeñas modificaciones heredadas, provechosas todas al ser conservado", escribió Darwin y dio nombre a este proceso (El origen de las especies, selección natural) [44].

Estas "modificaciones heredadas", señaladas por Darwin como las generadoras de organismos mejores, son llamadas mutaciones hoy en día y constituyen el corazón de la evolución. Un organismo mutante ha sufrido una modificación que lo hace diferente al resto de sus semejantes. Esta modificación puede ser un inconveniente para él, pero puede ocurrir también que le otorgue alguna cualidad que le permita sobrevivir más fácilmente que al resto de los individuos de su especie. Este organismo tendrá mayor probabilidad de reproducirse y heredará a sus descendientes la característica que le dio ventaja (supervivencia del más apto). Con el tiempo, gracias a la competencia, los organismos que en un principio eran raros se volverán comunes a costa de la desaparición de aquellos menos aptos, dando entonces un paso en el proceso evolutivo [44] (figura 3.1).

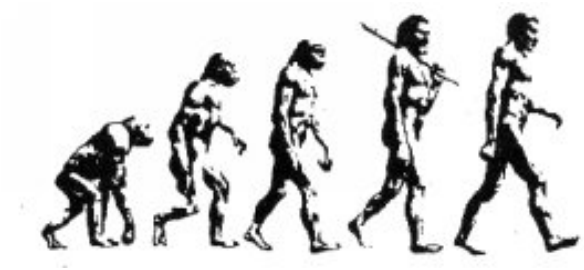


Figura 3.1: Evolución biológica

Por otro lado, un par de años más tarde, el monje austriaco Johann Gregor Mendel realizó una serie de experimentos con guisantes, estudiando las características básicas de esta planta. Gracias a este análisis, Mendel descubrió tres leyes básicas que gobernaban el paso de una característica de un miembro de una especie a otro. La primera ley (llamada de Segregación) establecía que los miembros de cada par de alelos de un gen se separan cuando se producen los gametos durante la meiosis. La segunda ley (llamada de la Independencia) establecía que los pares de alelos se independizan (o separan entre si) durante la formación de gametos. La tercera ley (llamada de la Uniformidad) establecía que cada características heredada se determina mediante dos factores provenientes de ambos padres, lo cual decide si un cierto gen es dominante o recesivo [11].

En base a estas dos teorías, entre los años 1920 y 1950, surge lo que hoy se denomina Neo-Darwinismo. El Neo-Darwinismo establece que la historia de la vida en nuestro planeta puede ser explicada a través de un conjunto de procesos estadísticos que actúan sobre y dentro de las poblaciones y especies [26]: la reproducción, la mutación y la selección.

El término "computación evolutiva" o "algoritmos evolutivos", realmente engloba una serie de técnicas inspiradas biológicamente en la teoría Neo-Darwiniana. Aunque hoy en día es cada vez más difícil distinguir las diferencias entre los distintos tipos de algoritmos evolutivos existentes, suele hablarse de tres paradigmas principales:

- Programación Evolutiva
- Estrategias Evolutivas
- Algoritmos Genéticos

Cada uno de estos paradigmas se originó de manera independiente y con motivaciones muy distintas [11].

Como se mencionó, este capítulo se concentrará principalmente en los algoritmos genéticos, tomando como primera fuente los apuntes de clase del Dr. Coello Coello [11].

Figura 3.2: Molécula de ADN

3.2. Terminología Biológica y Algoritmos Evolutivos

3.2.1. Fundamentos Biológicos

El ácido desoxirribonucleico (*ADN*) es la molécula portadora de toda la información genética que pasa de una generación a otra, y contiene todas las especificaciones para la formación de un organismo nuevo, como también para el control de todas las actividades de las células durante la vida del mismo. Este nuevo individuo puede ser idéntico a aquel del que proviene, o casi similar, en el caso de mezclarse con otra cadena (reproducción sexual) o de sufrir mutaciones. Por lo tanto, el ADN tiene como función principal la herencia, siendo el ADN una especie de plano o receta para las proteínas que conforman un organismo.

Cada molécula de ADN está constituida por dos cadenas o bandas formadas por un elevado número de compuestos químicos llamados nucleótidos. Estas cadenas forman una especie de escalera retorcida que se llama doble hélice (figura 3.2) y fue descubierta por James Watson y Francis Crick [53]. Cada nucleótido contiene un segmento de la estructura de soporte (azúcar + fosfato), que mantiene la cadena unida, y una base nitrogenada, que interacciona con la otra cadena de ADN en la hélice. En general, una base ligada a un azúcar se denomina nucleósido y una base ligada a un azúcar y a uno o más grupos fosfatos recibe el nombre de nucleótido. Cuando muchos nucleótidos se encuentran unidos, como ocurre en el ADN, el polímero resultante se denomina polinucleótido [46].

Se denomina *cromosoma* a una de las cadenas de ADN que se encuentra en el núcleo de las células. Los cromosomas son responsables de la transmisión de información genética.

Un *gen* es una sección de ADN que codifica una cierta función bioquímica definida, usualmente la producción de una proteína, y es fundamentalmente una unidad de herencia.

Cada gen es capaz de ocupar sólo una región en particular de un cromosoma. En cada determinado lugar pueden existir, en la población, formas alternativas del gen. A estas formas alternativas se les llama *alelos*.

Se dice que un gen es epistático cuando su presencia suprime el efecto de un gene que se encuentra en otra posición. Los genes epistáticos son llamados algunas veces genes de inhibición por el efecto que producen sobre otros genes.

Se llama *genoma* a la colección total de genes (y por tanto, cromosomas) que posee un organismo.

Una célula haploide es aquella que contiene la mitad del número normal de cromosomas (2n es el caso de una célula diploide). Un ejemplo de esto son las células reproductoras (*gametos*), como los óvulos y los espermatozoides de los mamíferos que contienen un sólo juego de cromosomas, mientras que el resto de las células de un organismo superior suelen tener un par de ellos. Cuando los gametos se unen durante la fecundación, el huevo fecundado contiene un número normal de cromosomas transformándose en una célula diploide.

Se denomina reproducción a la creación de un nuevo individuo a partir de:

- Dos progenitores (sexual)
- Un progenitor (asexual)

En la naturaleza, la mayoría de las especies capaces de reproducirse sexualmente son diploides. Durante la reproducción sexual ocurre la recombinación (o cruza):

- Caso Haploide: Se intercambian los genes entre los cromosomas (haploides) de los dos padres.
- Caso Diploide: En cada padre, se intercambian los genes entre cada par de cromosomas para formar un gameto, y posteriormente los gametos de los 2 padres se aparean para formar un solo conjunto de cromosomas diploides.

Durante la mutación, se cambian nucleótidos individuales de padre a hijo. La mayoría de estos cambios se producen por errores de copiado durante la reproducción (Figura 3.3).

Se denomina *individuo* a un solo miembro de una población. Se denomina Población a un grupo de individuos que pueden interactuar juntos, por ejemplo, para reproducirse.

Se denomina $fenotipo$ a los rasgos (observables) específicos de un individuo. Se denomina *genotipo* a la composición genética de un organismo (la información contenida en el genoma o conjunto de todos los cromosomas), es decir, es lo que potencialmente puede llegar a ser un individuo. El genotipo da origen, tras el desarrollo fetal y posterior, al fenotipo del organismo.

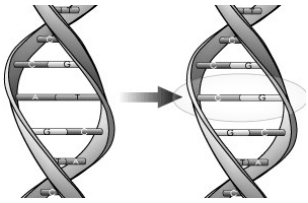


Figura 3.3: Mutación Celular

La *aptitud* de un individuo se define como la probabilidad de que éste viva para reproducirse (viabilidad), o como una función del número de descendientes que éste tiene (fertilidad).

Se denomina *ambiente* a todo aquello que rodea a un organismo. Puede ser "físico" (abiótico) o biótico. En ambos casos, el organismo ocupa un nicho que ejerce una influencia sobre su aptitud dentro del ambiente total. A través de varias generaciones, los ambientes bióticos pueden fomentar la co-evolución, en la cual la aptitud se determina mediante la selección parcial de otras especies.

La selección es el proceso mediante el cual algunos individuos en una población son seleccionados para reproducirse, típicamente con base en su aptitud. La selección dura se da cuando sólo los mejores individuos se mantienen para generar descendientes. La selección blanda se da cuando se usan mecanismos probabilísticos para mantener como padres a individuos que tengan aptitudes relativamente bajas.

Aunque no existe una definición totalmente aceptada de *especie*, se dirá que es una colección de criaturas vivientes que tienen características similares, y que se pueden reproducir entre sí.

Se denomina especiación al proceso mediante el cual aparece una especie. La causa más común de especiación es el aislamiento geográfico. Si una subpoblación de una cierta especie se separa geográficamente de la población principal durante un tiempo suficientemente largo, sus genes divergirán. Estas divergencias se deben a diferencias en la presión de selección en diferentes lugares, o al fenómeno conocido como desvío genético (cambios en las frecuencias de genes/alelos en una población con el paso de muchas generaciones, como resultado del azar en vez de la selección).

En los ecosistemas naturales, hay muchas formas diferentes en las que los animales pueden sobrevivir (en los árboles, de la cacería, en la tierra, etc.) y cada estrategia de supervivencia es llamada un nicho ecológico. Dos especies que ocupan nichos diferentes (por ejemplo, una que se alimenta de plantas y otra que se alimenta de insectos) pueden coexistir entre ellas sin competir, de una manera estable. Sin embargo, si dos especies que ocupan el mismo nicho se llevan a la misma zona, habrá competencia, y a la larga, la especie más débil se extinguirá (localmente). Por lo tanto, la diversidad de las especies

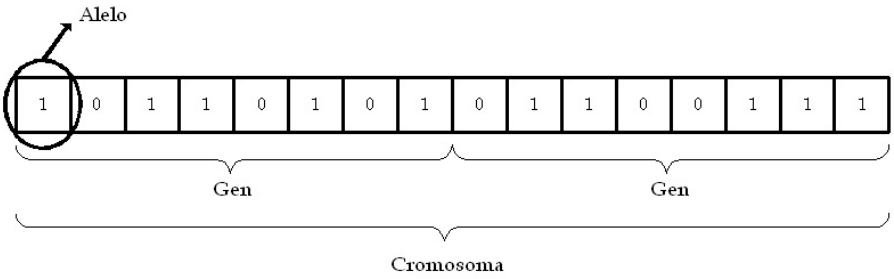


Figura 3.4: Cadena cromosómica

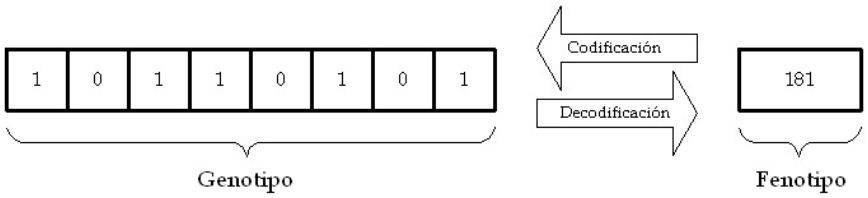


Figura 3.5: Ejemplo de codificación genotípica y decodificación fenotípica

depende de que ocupen una diversidad de nichos (o de que estén separadas geográficamente).

Se denomina migración a la transferencia de (los genes de) un individuo de una subpoblación a otra.

3.2.2. Conceptos de la Computación Evolutiva

Se denomina *cromosoma* a una estructura de datos que contiene una cadena de parámetros de diseño o genes. De esta manera, un *gen* es una subsección de un cromosoma que (usualmente) codifica el valor de un solo parámetro (por ejemplo, en codificación binaria) y se denomina *alelo* a cada valor posible que puede adquirir un gen (en codificación binaria sería un 0 o un 1) (figura 3.4).

Se denomina *genotipo* a la codificación total de los parámetros que representan una solución del problema a resolverse, mientras que la decodificación del cromosoma se denomina $fenotipo$ (figura 3.5).

Se denomina *individuo* a un solo miembro de la población de soluciones potenciales a un problema y contiene un cromosoma (o genoma) que lo representa.

Se denomina *aptitud* al valor que se asigna a cada individuo y que indica qué tan bueno es éste con respecto a los demás para la solución de un problema. Como una visión más global, se denomina *paisaje de aptitud* (fitness landscape) a la hipersuperficie que se obtiene al aplicar la función de aptitud a cada punto del espacio de búsqueda.

Se llama generación a una iteración de la medida de aptitud y a la creación de una nueva población por medio de operadores de reproducción.

Una población puede subdividirse en grupos a los que se denomina *subpoblaciones*. Normalmente, sólo pueden cruzarse entre sí los individuos que pertenezcan a la misma subpoblación, emulando de esta manera el fenómeno natural de Especiación. En estos esquemas, suele permitirse la Migración de individuos de una subpoblación a otra.

Hay un tipo de población usada en computación evolutiva en la que cualquier individuo puede reproducirse con otro con una probabilidad que depende sólo de su aptitud. A esta clase de poblaciones se las denomina Poblaciones Panmíticas y la mayor parte de los algoritmos evolutivos las utilizan. Como contrapartida, puede decirse que lo opuesto a estas poblaciones es permitir la reproducción sólo entre individuos de la misma subpoblación.

Debido a ruidos estocásticos, los algoritmos evolutivos tienden a converger a una sola solución. Para evitar eso, y mantener la diversidad, existen técnicas que permiten crear distintos *nichos* para los individuos.

Se llama *bloque constructor* a un grupo pequeño y compacto de genes que han coevolucionado de tal forma que su introducción en cualquier cromosoma tiene una alta probabilidad de incrementar la aptitud de dicho cromosoma.

Se llama decepción a la condición donde la combinación de buenos bloques constructores llevan a una reducción de aptitud, en vez de un incremento, para algunos problemas [25].

Se llama epístasis a la interacción entre los diferentes genes de un cromosoma, es decir, a la medida en que la contribución de aptitud de un gene depende de los valores de los otros genes. Cuando un problema tiene poca epístasis (o ninguna), su solución es trivial (un algoritmo escalando la colina es suficiente para resolverlo). Cuando un problema tiene una epístasis elevada, el problema será deceptivo, por lo que será muy difícil de resolver por un algoritmo evolutivo.

Se denomina *esquema* a un patrón de valores de genes de un cromosoma que puede incluir estados "no interesa". Por ejemplo, usando un alfabeto binario, los esquemas se forman del alfabeto $0, 1, \#$, por lo que el cromosoma 0110 es una instancia del esquema $\#1\#0$ (donde $\#$ significa "no interesa").

Se llama operador de reproducción a todo aquel mecanismo que influencia la forma en que se pasa la información genética de padres a hijos. Los operadores de reproducción caen en tres amplias categorías:

- Cruza
- Mutación
- Reordenamiento

La *cruza* es un operador que forma un nuevo cromosoma combinando partes de cada uno de sus cromosomas padres.

Se denomina mutación a un operador que forma un nuevo cromosoma a través de alteraciones (usualmente pequeñas) de los valores de los genes de un solo cromosoma padre.

Un operador de reordenamiento es aquél que cambia el orden de los genes de un cromosoma, con la esperanza de juntar los genes que se encuentren relacionados, facilitando así la producción de bloques constructores. La inversión es un ejemplo de un operador de reordenamiento en el que se invierte el orden de todos los genes comprendidos entre 2 puntos seleccionados al azar en el cromosoma.

Es importante aclarar que en los algoritmos genéticos los operadores de reproducción actúan sobre los genotipos y no sobre los fenotipos de los individuos.

Se denomina *elitismo* al mecanismo utilizado en algunos algoritmos evolutivos para asegurar que los cromosomas de los miembros más aptos de una población se pasen a la siguiente generación sin ser alterados por ningún operador genético. Usar elitismo asegura que la aptitud máxima de la población nunca se reducirá de una generación a la siguiente. Sin embargo, no necesariamente mejora la posibilidad de localizar el óptimo global de una función. No obstante, es importante hacer notar que se ha demostrado que el uso de elitismo es vital para poder demostrar convergencia de un algoritmo genético [48].

Cuando se atraviesa un espacio de búsqueda, se denomina explotación al proceso de usar la información obtenida de los puntos visitados previamente para determinar qué lugares resulta más conveniente visitar a continuación. Se denomina exploración al proceso de visitar completamente nuevas regiones del espacio de búsqueda, para ver si puede encontrarse algo prometedor. La exploración involucra grandes saltos hacia lo desconocido y es buena para evitar quedar atrapado en óptimos locales. La explotación normalmente involucra movimientos finos y es buena para encontrar óptimos locales.

Generar la población inicial;
$CrearPoblacion(N)$;
while (la población no ha convergido) do
  Seleccionar individuos para reproducción;
  Crear descendencia aplicando recombinación y/o mutación a los individuos seleccionados;
  Computar la aptitud de los nuevos individuos;
  Eliminar a los individuos antiguos para hacer lugar para los nuevos cromosomas;
  Insertar la descendencia en la nueva generación;
end while
Salida : la mejor solución encontrada;

Figura 3.6: Algoritmo Genético Básico

3.3. Algoritmo Genético Básico

Los algoritmos genéticos fueron desarrollados por John H. Holland a principios de los años 1960 [27, 28], motivado por resolver problemas dentro del entorno de aprendizaje de máquina. Este algoritmo enfatiza la importancia de la cruza sexual (operador principal) sobre el de la mutación (operador secundario), y usa selección probabilística.

Un algoritmo genético consiste en hallar los parámetros de los cuales depende el problema, codificarlos en una representación específica, y aplicar los métodos de evolución: selección, reproducción sexual y mutaciones que generan diversidad (la figura 3.6 muestra el algoritmo propiamente dicho).

Para poder aplicar el algoritmo genético se requiere de los 5 componentes básicos siguientes:

- Una representación de las posibles soluciones del problema.
- Una forma de crear una población inicial de posibles soluciones.
- Una función de evaluación que clasifique las soluciones en términos de su "aptitud" a sobrevivir en el ambiente.
- Operadores genéticos que permitan crear los hijos de las próximas generaciones.
- Valores para los diferentes parámetros que utiliza el algoritmo genético (tamaño de la población, probabilidad de cruza, probabilidad de mutación, número máximo de generaciones, etc.)

3.4. Representación de Parámetros

Los algoritmos genéticos no presentan restricciones en cuanto a la forma que deben adoptar los genes. Estos pueden representarse como cadenas binarias, números enteros

o reales, u otras estructuras más complejas, siempre considerando que la codificación seleccionada refleje de manera clara las similitudes entre individuos [40].

A continuación se detallan los métodos de codificación básicos.

3.4.1. Codificación Binaria

En la codificación binaria cada gen del cromosoma es codificado usando el mismo número de bits y cada posición puede tomar el valor 0 o 1. La precisión de esta aproximación depende, para un dominio de tamaño fijo, del número de bits que se utilizará para representar cada gen y es igual a

$$Presicion = \frac{(LS - LI)}{(2^n - 1)} \tag{3.1}$$

donde LI y LS son los límites inferior y superior, respectivamente, del dominio de los parámetros y n es el número de bits utilizados para representar cada gen del cromosoma (longitud de la palabra) [42].

Por lo general, este tipo de codificación se utiliza para problemas cuyos parámetros poseen dos estados posibles (Si/No, Aprobado/Desaprobado, etc.), dando a un estado el valor 0 y al otro el valor 1 [40], aunque también es utilizada para codificar números enteros y reales (por ejemplo, con la representación de la IEEE para precisión simple de 32 bits). Normalmente, la codificación es estática, pero en casos de optimización numérica, el número de bits dedicados a codificar un parámetro puede variar, o incluso los que representen los bits dedicados a codificar cada parámetro.

Argumentando la utilización de codificación binaria en los algoritmos genéticos, Holland [29] comparó dos representaciones diferentes que tuvieran aproximadamente la misma capacidad de acarreo de información, pero de entre ellas, una tenía pocos alelos y cadenas largas (por ejemplo, cadenas binarias de 80 bits de longitud) y la otra tenía un número elevado de alelos y cadenas cortas (por ejemplo, cadenas decimales de longitud 24). Nótese que 2^{80} se aproxima a 10^{24}. Holland [29] argumentó que la primera codificación da pie a un grado más elevado de "paralelismo implícito" porque permite más "esquemas" que la segunda.

El hecho de contar con más esquemas favorece la diversidad e incrementa la probabilidad de que se formen buenos "bloques constructores" (es decir, la porción de un cromosoma que le produce una aptitud elevada a la cadena en la cual está presente) en cada generación, lo que en consecuencia mejora el desempeño del algoritmo genético con el paso del tiempo de acuerdo al teorema de los esquemas [25, 29, 11].

Finalmente, esta representación, dependiendo de la complejidad del problema, posee algunas desventajas fácilmente observables:

- El cromosoma puede implicar una longitud demasiado grande.
- Los operadores genéticos deberán ser adaptados para cada tipo de problema, ya que pueden ocasionar soluciones ilegales.
- No siempre dos cadenas binarias representan en buena medida la distancia real de los fenotipos a los que codifican (medido en base a la "Distancia de Hamming" [1]).
- No brinda una representación natural (por ejemplo, para el problema del viajero).
- Posee un alto grado de epístasis.

Estas desventajas hacen que la codificación binaria simple no sea la representación más adecuada, siendo más ventajoso utilizar la codificación Gray que se detalla en la sección próxima.

3.4.2. Codificación Gray

Como se dijo anteriormente, la representación binaria no mapea adecuadamente el espacio de búsqueda con el espacio de representación. Por ejemplo, si se codifica en binario a los números 1 y 2, los cuales están adyacentes en el espacio de búsqueda, sus equivalentes en binario, 001 y 010, tienen una "Distancia de Hamming" de dos bits en el espacio de representación. Ante este problema es dónde se utiliza la codificación Gray, ya que asegura que la propiedad de adyacencia en el espacio de búsqueda se conserve en el espacio de representación.

La codificación Gray de un número binario se realiza de la siguiente manera:

- El primer bit (el más significativo) se define idéntico al del número que se está codificando.
- Los bits siguientes se obtienen realizando un XOR de sus bits consecutivos.

Por ejemplo:

- $i = 0 \rightarrow binario = 0000 \rightarrow gray = 0000$.
- $i = 1 \rightarrow binario = 0001 \rightarrow gray = 0001$.
- $i = 2 \rightarrow binario = 0010 \rightarrow gray = 0011$.
- $i = 3 \rightarrow binario = 0011 \rightarrow gray = 0010$.
- $i = 4 \rightarrow binario = 0100 \rightarrow gray = 0110$.

[1] La "Distancia de Hamming" entre dos cadenas binarias es la cantidad de posiciones en que ambas difieren

En resumen, la representación Gray tiene la propiedad de que dos números consecutivos sólo difieren en 1 bit en el espacio de representación. También se ha demostrado empíricamente que el uso de códigos Gray mejora el desempeño del algoritmo genético al aplicarse a las funciones de prueba clásica de De Jong [31].

3.4.3. Codificación de Números Reales

Los códigos de Gray son útiles para representar enteros, pero su desventaja se presenta cuando se trabaja con un espacio de búsqueda contínuo, ya que sólo un subconjunto de estos podría representarse computacionalmente. La solución consiste en acotar el intervalo de interés, muestrearlo con cierta precisión, y representar este conjunto discreto de puntos con una codificación entera.

Si se utilizan b bits para representar un parámetro x, tal que $x \in [x_{min}, x_{max}]$, entonces la presición está dada por la distancia d entre dos muestras consecutivas de x [40] (ecuación 3.2).

$$d = \frac{(x_{max} - x_{min})}{(2^b)} \tag{3.2}$$

Entonces, los valores que x puede tomar pertenecen al conjunto discretizado $\{c_{min}, c_{min} + d, c_{min} + 2d, ..., c_{max}\}$.

De la definición se desprende que $d \to 0$ cuando $b \to \infty$. Por lo tanto, aumentar la cantidad b de bits para representar el parámetro x permite que la presición aumente y que x pueda tomar un mayor número de valores.

El problema más importante que se debe resolver para muestrear números reales consiste en determinar la precisión adecuada. Por ejemplo, si se tiene una variable entre 0,35 y 1,40 con una precisión de 2 decimales requiere $\log_2 140 - 35 = 7$ bits para representar cualquier real en ese rango. Note que el problema continúa porque el número 0,38 se representaría como 0000011 mientras que el número 0,39 sería 0000101. Aunque se utilicen códigos Gray, si se tienen demasiadas variables y se busca una buena precisión, la dimensión del cromosoma será alta y el algoritmo genético tenderá a tener un desempeño pobre.

Una alternativa podría ser el uso de un formato binario estándar para representar números reales (la representación de la IEEE para precisión simple de 32 bits). Esto permite representar un rango grande de números reales usando una cantidad fija de bits, pero tiene los problemas antes mencionados de la representación binaria estándar.

Mientras los teóricos afirman que los alfabetos pequeños son más efectivos que los alfabetos grandes, los prácticos demostraron a través de una cantidad significativa de aplicaciones del mundo real (particularmente problemas de optimización numérica) que el uso directo de números reales en un cromosoma funciona mejor en la práctica que la representación binaria tradicional [14, 37], donde la mutación tiene un lugar fundamental.

Figura 3.7: Una representación entera de números reales

Sin embargo, los teóricos de los algoritmos genéticos han criticado fuertemente el uso de valores reales en los genes de un cromosoma, principalmente porque esta representación tiende a hacer que el comportamiento del algoritmo genético sea más errático e impredecible. Debido a esto, se han diseñado varios operadores especiales para emular el efecto de la cruza y la mutación en los alfabetos binarios [56, 37, 14, 11].

También se han utilizado otras representaciones de los números reales. Por ejemplo, el uso de enteros para representar cada dígito fue aplicado exitosamente a varios problemas de optimización [12, 13]. La figura 3.7 muestra un ejemplo de la representación del número 1.45679 usando enteros. En este caso, la posición del punto decimal en cada variable es fija, y puede variar de variable en variable codificada en el cromosoma. La precisión está limitada por la longitud de la cadena, y puede incrementarse o decrementarse según se desee. Los operadores de cruza tradicionales (un punto, dos puntos y uniforme) pueden usarse directamente en esta representación, y la mutación puede consistir en generar un dígito aleatorio para una cierta posición o bien en producir una pequeña perturbación (por ejemplo, ± 1) para evitar saltos extremadamente grandes en el espacio de búsqueda. Esta representación pretende mantener lo mejor de la codificación binaria y real al incrementar la cardinalidad del alfabeto utilizado, pero manteniendo el uso de los operadores genéticos tradicionales casi sin cambios.

3.5. Función de Aptitud

En la naturaleza hay individuos más aptos que otros para sobrevivir. Todas las características que cada individuo posea señalan alguna diferencia entre los individuos, siendo esta diferencia relativa al resto de la población a la que pertenece. De manera similar, en los algoritmos genéticos es necesario establecer algún criterio que permita decidir cuáles de las soluciones propuestas en una población son mejores respecto del resto de las propuestas y cuáles no lo son. Es necesario establecer, para cada individuo, una medida de desempeño relativa a la población a la que pertenece [44].

Un buen diseño de la función de aptitud (también conocida como función objetivo o función de fitness) resulta extremadamente importante para el correcto funcionamiento de un algoritmo genético. Esta función permite cuantificar el grado de aproximación de cada individuo a la solución del problema y, por lo tanto, permite distinguir a los mejores individuos de los peores. Esta calificación está dada por un número real no negativo que, cuanto mejor sea la solución propuesta por dicho individuo, mayor va a ser su valor. El objetivo de este número es que permita distinguir propuestas de solución buenas de aquéllas que no lo son.

Uno de los problemas de los algoritmos genéticos es su incapacidad de abordar restricciones. Idealmente, la representación debería ser tal que solo se pudieran generar soluciones factibles. Si todos los cromosomas posibles corresponden a soluciones factibles, entonces no hay necesidad de imponer restricciones, pero para que esto ocurra se supone que la función de aptitud debe ser capaz de tomar en cuenta las violaciones a las restricciones. El enfoque clásico a este caso es el uso de la función de penalidad que evalúa la magnitud de la violación. Esta función de penalidad se usa de manera integrada en la función de aptitud de manera de disminuir su valor en base a las violaciones que comete el individuo evaluado. Por lo tanto, si la penalidad es lo suficientemente grande, individuos altamente inviables raramente serán seleccionados para la selección, y el algoritmo genético se concentrará en soluciones factibles [20].

Existen ciertos problemas, en dónde la evaluación de cada individuo implica un costo computacional demasiado elevado, ya que la función de aptitud se evalúa una vez en cada individuo durante cada generación. Para casos en dónde la función es muy costosa generalmente se implementa una función de evaluación aproximada.

3.6. Población

Existen dos cuestiones básicas a tener en cuenta acerca de la población utilizada en un algoritmo genético: su inicialización y su tamaño. Estos dos aspectos son fundamentales para la diversidad genética de la población y, por consiguiente, para el comportamiento del algoritmo.

Conceptualmente, la primera población debería tener un contenido genético tan amplio como sea posible (deberían estar presentes todos los distintos alelos existentes de cada gen) para que pueda explorar la totalidad del espacio de búsqueda y converger. Para alcanzar este objetivo, en la mayoría de los casos se elige a la primera población de forma aleatoria, aunque a veces se puede usar alguna clase de heurística para establecer la población inicial. Esta condición de diversidad, no solo es necesaria en la población inicial, sino también en las generaciones siguientes, ya que brinda una aptitud promedio de la población más alta y agiliza la ejecución del algoritmo [20].

En cuanto al tamaño de la población, cuanto mayor sea, más fácil es explorar el espacio de búsqueda. Como restricción evidente, se sabe que el tiempo requerido por un algoritmo genético para converger está dado por funciones de evaluación del orden $(n \log n)$ donde n es el tamaño de la población [24], aunque por otro lado, se ha demostrado que la eficiencia de un algoritmo genético para alcanzar un óptimo global está determinada en gran medida por el tamaño de la población. De esta manera, una población grande es muy útil pero requiere un costo computacional mayor, por lo que habría que analizar la relación tamaño/performance que brinde soluciones más óptimas (por ejemplo, en la práctica, poblaciones de 20 individuos han demostrado ser una alternativa válida para este problema) [20].

3.7. Métodos de Selección

Los individuos elegidos con el operador de selección aportarán sus genes a la población siguiente, por lo tanto, es deseable que el operador de selección elija los mejores individuos de la población actual, siempre manteniendo la diversidad y mejorando o igualando el fitness en la población siguiente. En el algoritmo genético este proceso de selección suele realizarse de forma probabilística (es decir, aún los individuos menos aptos tienen una cierta oportunidad de sobrevivir), a diferencia de las estrategias evolutivas, en las que la selección es *extintiva* (los menos aptos tienen cero probabilidades de sobrevivir).

Las técnicas de selección usadas en algoritmos genéticos pueden clasificarse en tres grandes grupos:

- Selección proporcional
- Selección mediante torneo
- Selección de estado uniforme

3.7.1. Selección Proporcional

Este grupo describe una serie de técnicas que eligen individuos de acuerdo a su contribución de aptitud con respecto al total de la población [29].

La Ruleta

La Selección por Ruleta es la más popular de las selecciones proporcionales. En esta técnica, se asigna a cada solución un sector de la ruleta cuyo tamaño o ángulo es proporcional a la medida de aptitud que posea. Entonces se elige una posición al azar sobre la ruleta (es de decir, se gira la ruleta), y de esta manera se selecciona la solución asignada a la posición elegida. El algoritmo es simple, pero ineficiente (su complejidad es $O(n^2)$). Asimismo, presenta el problema de que el individuo menos apto puede ser seleccionado más de una vez, provocando así una diferencia entre el valor esperado y el real de descendientes.

En la figura 3.8 contiene el pseudocódigo del algoritmo de selección basado en la Técnica de la Ruleta plateada por De Jong [31]. La figura 3.9 muestra un ejemplo del método.

Sobrante Estocástico

Esta técnica de selección fue propuesta por Booker [7] y Brindle [8] como una alternativa para aproximarse más a los valores esperados ($ValEsp$) de los individuos:

$$ValEsp_i = \frac{ValordeFitness_i}{FitnessPromedio} \tag{3.3}$$

CalcularSumatoriaDelValorEsperadoDeLaPoblacion(T);
for (i=1 to TamañoDeLaPoblación) do
    Generar un número aleatorio r entre 0 y T;
    Recorrer secuencialmente los individuos, sumando los valores esperados,
    hasta que la suma sea mayor o igual a r;
    El individuo que haga que la suma exceda el límite r es el seleccionado;
end for

Figura 3.8: Algoritmo basado en la Técnica de la Ruleta

La idea principal es asignar determinísticamente las partes enteras de los valores esperados para cada individuo y luego usar otro esquema (proporcional) para la parte fraccionaria.

Hay 2 variantes principales de este método:

- *Sin Reemplazo*: Cada sobrante se usa para sesgar el tiro de una moneda que determina si una cadena se selecciona de nuevo o no (Complejidad $O(n)$).
- *Con Reemplazo*: Los sobrantes se usan para dimensionar los segmentos de una ruleta y se usa esta técnica de manera tradicional (Complejidad $O(n^2)$).

La más popular es la versión sin reemplazo, la cual parece ser superior a la ruleta [7, 11].

El problema de este método, si bien reduce los problemas de la ruleta, es que puede causar convergencia prematura al introducir una mayor presión de selección.

En la figura 3.10 contiene el pseudocódigo del algoritmo de selección basado en la Técnica del Sobrante Estocástico plateada en [7, 8]. La figura 3.11 muestra un ejemplo del método.

Universal Estocástica

Esta técnica fue propuesta por Baker [3] con el objetivo de minimizar la mala distribución de los individuos en la población en función de sus valores esperados. En la figura 3.12 puede verse el algoritmo de selección basado en la Técnica Universal Estocástica. La figura 3.13 muestra un ejemplo del método.

Aunque esta técnica tiene una complejidad de $O(n)$ posee los siguientes problemas:

- Puede ocasionar convergencia prematura.
- Hace que los individuos más aptos se multipliquen muy rápidamente.

Individuo	Fitness	Prob. De Selección	Valor Esperado = Número de Descendientes Esperado
Individuo 1	0,25	10%	0,5859375
Individuo 2	0,1	4%	0,234375
Individuo 3	0,36	14%	0,84375
Individuo 4	0,7	27%	1,640625
Individuo 5	0,65	25%	1,5234375
Individuo 6	0,5	20%	1,171875
Sumatoria	**2,56**	**1**	**6**
Promedio	**0,426666667**		

Valor	Cálculo
Probabilidad de Selección	Valor de fitness/Suma de fitness
Número de Descendientes Esperados	Valor de fitness/Fitness Promedio

Selección
En este ejemplo T = 6 y suponga que **r** = 1.3
Hasta el individuo 2 la suma es 0,8203125 < r
Al sumar el individuo 3 la suma es 1,6640625 > r
Por lo tanto, se selecciona al individuo 3

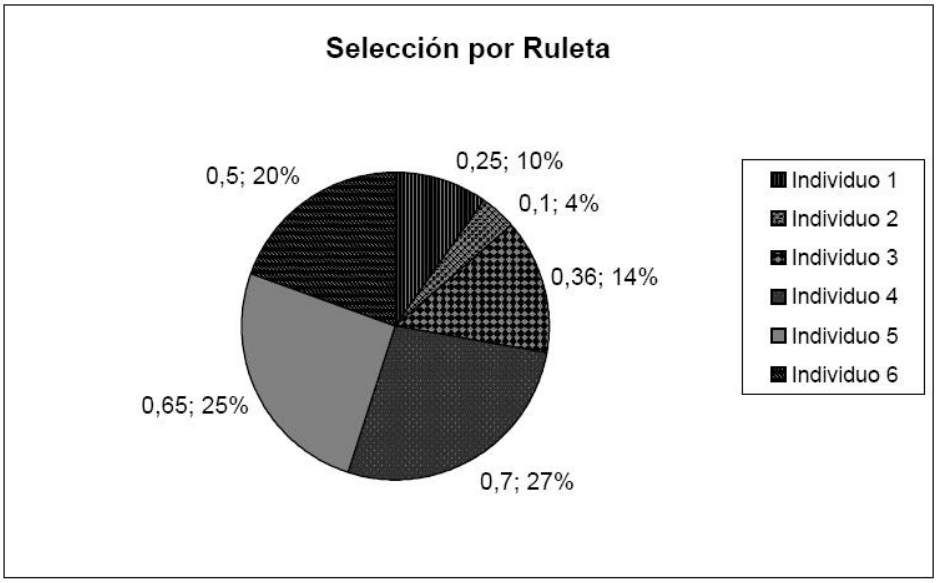


Figura 3.9: Ejemplo de selección mediante la Técnica de la Ruleta

Asignar de determinísticamente la cantidad de valores esperados a cada individuo (parte entera);
Usar probabilísticamente los sobrantes del redondeo para rellenar la población;

Figura 3.10: Algoritmo basado en la Técnica del Sobrantes Estocástico

Individuo	Fitness	ValEsp	Parte Entera	Dif	Padres iniciales
Individuo 1	0,25	0,5859375	0	0,5859375	No
Individuo 2	0,1	0,234375	0	0,234375	No
Individuo 3	0,36	0,84375	0	0,84375	No
Individuo 4	0,7	1,640625	1	0,640,625	Si
Individuo 5	0,65	1,5234375	1	0,5234375	Si
Individuo 6	0,5	1,171875	1	0,171875	Si
Sumatoria	**2,56**	**6**	**3**		
Promedio	**0,426666667**				

Método	Explicación	Parejas para los 3 padres Seleccionados	Parejas Formadas
Sin Reemplazo	Aplicar funcion flip(prob) hasta obtener el número de parejas necesarios, donde flip(prob) = true con probabilidad prob	flip(0,5859375) = True -> Individuo 1 flip(0,234375) = False -> Individuo 2 flip(0,84375) = True -> Individuo 3 ... flip(0,171875) = True -> Individuo 6	Individuo 4 e Individuo 1 Individuo 5 e Individuo 3 Individuo 6 e Individuo 6
Con Reemplazo	Armar ruleta tradicional con los sobrantes y tirar hasta obtener el número de parejas necesario	Tres tiros para los cuales se seleccionaron los individuos 1,3 y 4	Individuo 4 e Individuo 1 Individuo 5 e Individuo 3 Individuo 6 e Individuo 4

Figura 3.11: Ejemplo de selección mediante la Técnica del Sobrante Estocástico

$ptr = rand()$;
$sum = 0$;
i= 1;
while (i $\leq$ n) do begin
 sum =sum+$ValorEsperado(i)$;
 while (sum > ptr) do begin
 Seleccionar(i);
 ptr = ptr + 1;
 end;
 i=i+1;
end while

Figura 3.12: Algoritmo basado en la Técnica Universal Estocástica

Individuo	Fitness	Prob. De Selección	Valor Esperado = Número de Descendientes Esperado
Individuo 1	0,25	10%	0,5859375
Individuo 2	0,1	4%	0,234375
Individuo 3	0,36	14%	0,84375
Individuo 4	0,7	27%	1,640625
Individuo 5	0,65	25%	1,5234375
Individuo 6	0,5	20%	1,171875
Sumatoria	2,56	1	6
Promedio	0,426666667		

Selección
En este ejemplo T = 6 y suponga que ptr = 0,8
i=1---> sum=0,5859375 0,5859375 < ptr Termina ciclo interno
i=2---> sum=sum + 0,234375 = 0,8203125 0,5859375 > ptr Seleccionar Inividuo 2 ptr = ptr + 1 = 1,8 Termina ciclo interno
i=3---> sum=sum+0,84375 = 1,6640625 1,6640625 < ptr Termina ciclo interno
i=4---> sum=sum + 1,640625 = 3,304687 3,304687 > ptr Seleccionar Inividuo 4 ptr = ptr + 1 = 2,8 3,304687 > ptr Seleccionar Inividuo 4 ptr = ptr + 1 = 3,8 Termina ciclo interno
i=5---> sum=sum + 1,5234375 = 4,828125 4,828125 > ptr Seleccionar Inividuo 5 ptr = ptr + 1 = 4,8 4,828125 > ptr Seleccionar Inividuo 5 ptr = ptr + 1 = 5,8 Termina ciclo interno
i=2---> sum=sum + 1,171875 = 6 6 > ptr Seleccionar Inividuo 6 ptr = ptr + 1 = 6,8 Termina ciclo interno
Padres = {2,4,4,5,5,6}

Figura 3.13: Ejemplo de selección mediante la Técnica Universal Estocástica

Calcular la Probabilidad de Selección de todos los individuos de la población;
Calcular el valor esperado de cada individuo como $ValEsp_i$ = Probabilidad de Selección * n
Asignar determinísticamente la parte entera de $ValEsp_i$
Ordenar la población según los valores decimales
Obtener los padres faltantes de la parte superior de la lista

Figura 3.14: Algoritmo basado en la Técnica de Muestreo Determinístico

Individuo	Fitness	Probabilidad de Selección	ValEsp	Parte Entera	Dif	Orden
Individuo 1	0,25	0,0976563	0,5859375	0	0,5859375	3
Individuo 2	0,1	0,0390625	0,2343750	0	0,2343750	5
Individuo 3	0,36	0,1406250	0,8437500	0	0,8437500	1
Individuo 4	0,7	0,2734375	1,6406250	1	0,6406250	2
Individuo 5	0,65	0,2539063	1,5234375	1	0,5234375	4
Individuo 6	0,5	0,1953125	1,1718750	1	0,1718750	6
Sumatoria	2,56	1	6			
Promedio	0,426666667					

Padres	Individuo 4 e Individuo 3 Individuo 5 e Individuo 4 Individuo 6 e Individuo 1

Figura 3.15: Ejemplo de selección mediante la Técnica de Muestreo Determinístico

- No resuelve el problema más serio de la selección proporcional (o sea, la imprecisión entre los valores esperados y los números de copias de cada individuo que son realmente seleccionados).

Muestreo Determinístico

Esta es una variante similar al sobrante estocástico, pero requiere de un algoritmo de ordenación. En la figura 3.14 puede verse el algoritmo de selección y la figura 3.15 muestra un ejemplo del método.

Este algoritmo tiene una complejidad de $O(n)$ para la asignación determinística y es $O(n \log n)$ para la ordenación. En cuanto a los problemas que presenta, de manera evidente, son los mismos que los que posee la Técnica de Sobrante Estocástico.

3.7.2. Selección Mediante Torneo

Esta técnica fue propuesta por Wetzel [54]. La idea es hacer la selección con base en comparaciones directas entre los individuos. Este tipo de selección se puede hacer de manera determinista o de manera probabilística. En la figura 3.16 puede verse el algoritmo de selección determinístico y la figura 3.17 muestra un ejemplo del método.

Barajar los individuos de la población;
Escoger un número p de individuos (generalmente 2)
Comprarlos en base a su aptitud
El ganador del "torneo" es el individuo más apto
Debe barajarse la población un total de p veces para seleccionar N padres (N=tamaño de la población)

Figura 3.16: Algoritmo basado en la Técnica Mediante Torneo Determinístico

Orden	Individuo	Fitness	Barajar 1	Padres 1	Barajar 2	Padres 1
1	Individuo 1	0,25	3	3	5	5
2	Individuo 2	0,1	1		1	
3	Individuo 3	0,36	2		4	4
4	Individuo 4	0,7	5	5	6	
5	Individuo 5	0,65	4	4	3	3
6	Individuo 6	0,5	6		2	
Padres	Individuo 3 e Individuo 5 Individuo 5 e Individuo 4 Individuo 4 e Individuo 3					

Figura 3.17: Ejemplo de selección mediante la Técnica Mediante Torneo

El algoritmo de la versión probabilística es idéntico al anterior, excepto por el paso en que se escoge al ganador. En vez de seleccionar siempre al individuo con aptitud más alta, se aplica $flip(p)$ y si el resultado es cierto, se selecciona al más apto. De lo contrario, se selecciona al menos apto. El valor de p es fijo durante todo el proceso evolutivo y $0{,}5 \leq p \leq 1$. Observe que si $p = 1$, la técnica se reduce a la versión determinística.

La versión determinística garantiza que el mejor individuo será seleccionado p veces (siendo p el tamaño del torneo).

Este técnica es eficiente y fácil del implementar con una complejidad $O(n)$. Puede introducir una presión selectiva muy alta en la versión determinística ya que lo malos individuos no tienen oportunidad de sobrevivir y puede regularse variando el tamaño del torneo, a medida que se aumente el tamaño del torneo se realizará mayor presión selectiva.

3.7.3. Selección de Estado Uniforme

Esta técnica fue propuesta por Whitley [55] y se utiliza en los algoritmos genéticos no generacionales, en los cuales sólo unos cuantos individuos son reemplazados en cada generación (los menos aptos). Esta técnica suele usarse cuando se evolucionan sistemas basados en reglas (por ejemplo, sistemas de clasificadores) en los que el aprendizaje es incremental, está especializada en la selección y posee una complejidad de $O(n \log n)$. La figura 3.18 muestra el pseudocódigo del algoritmo.

Sea P la población original;
Seleccionar R individuos ($1 \leq R < M$) de entre los más aptos (ej: R=2);
Efectuar cruza y mutación a los R individuos seleccionados. Sean H sus hijos;
Elegir al mejor individuo en H (o a los μ mejores);
Reemplazar los μ peores individuos de G por los μ mejores individuos de H;

Figura 3.18: Algoritmo basado en la Técnica Mediante Estado Uniforme

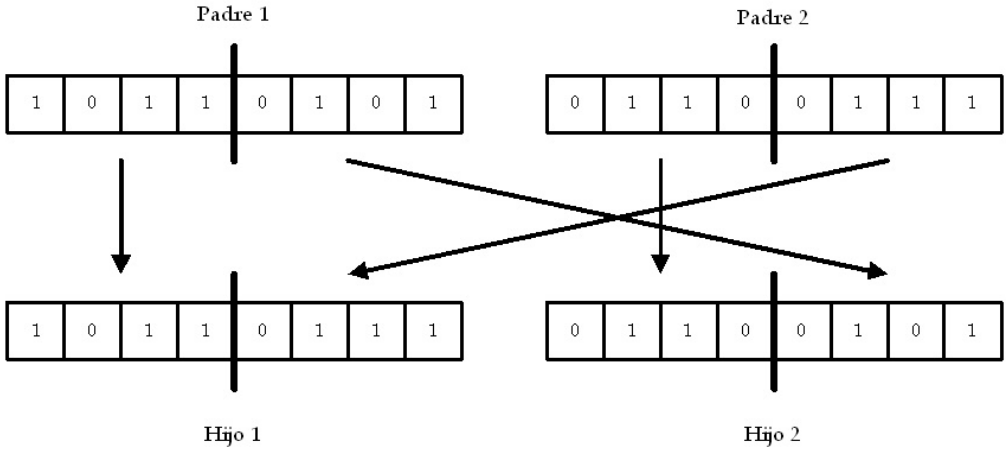


Figura 3.19: Ejemplo de Cruza de un Punto

3.8. Métodos de Cruce

El proceso de cruza consiste básicamente en seleccionar bloques de los cromosomas de individuos e intercambiarlos entre ellos para generar nuevos descendientes. Los efectos que la cruza tiene sobre la población son directos, ya que permite generar nuevas instancias de esquemas ya presentes en la población, como así también la creación de otros nuevos.

3.8.1. Cruza de un Punto

Esta técnica fue propuesta por Holland y, como su nombre lo dice, consiste en generar un punto P de manera aleatoria, que nos indica la posición hasta donde se intercambiarán los genes de los cromosomas. Como se muestra en la figura 3.19 la primera parte del cromosoma del hijo 1 está formada por la primera parte del padre 1 y la segunda parte está formada por la segunda parte del padre 2 y la primera parte del cromosoma del hijo 2 está formada por la primera parte del padre 2 y la segunda parte está formada por la segunda parte del padre 1.

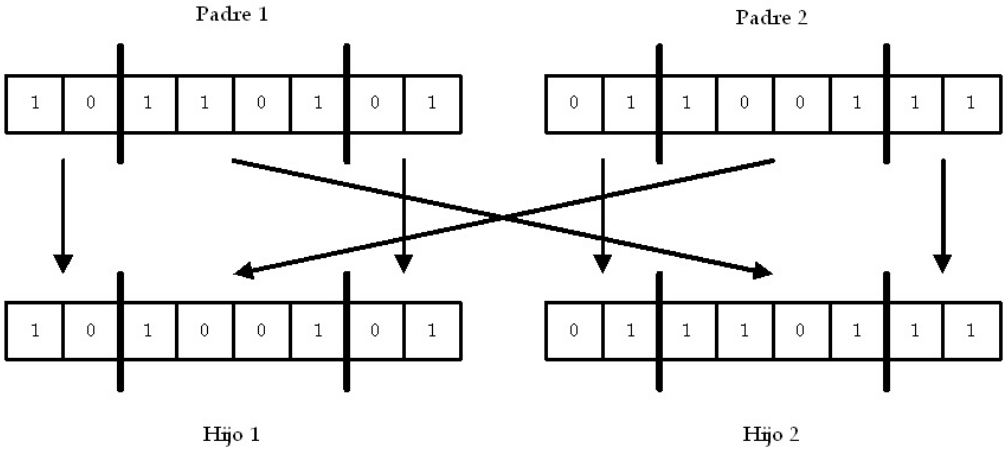


Figura 3.20: Ejemplo de Cruza de dos Puntos

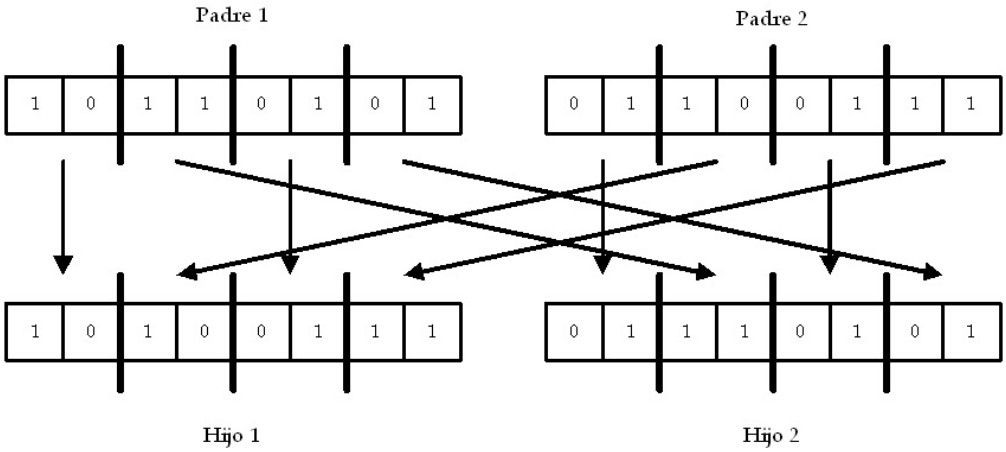


Figura 3.21: Ejemplo de Cruza Uniforme

3.8.2. Cruza de dos Puntos

En este caso, la cruza de 2 puntos consiste en elegir de manera aleatoria dos puntos, los cuales indicarán los bloques que se intercambiarán para producir a los hijos, como se ilustra en la figura 3.20.

3.8.3. Cruza Uniforme

Este tipo de cruza puede considerarse como una cruza generalizada, ya que puede tener n puntos de cruza. Para ello se utiliza un valor Pc el cual indica la probabilidad de tomar un gen de un padre o de otro. Si se elige $Pc = 0{,}5$, los hijos deben de estar formados por exactamente la mitad de cada padre, como se muestra en la figura 3.21.

3.8.4. Otros tipos de cruza

Existen otros tipos de cruza que se han utilizado en otros trabajos, tal como es el caso de la cruza acentuada [50] en un intento por implementar un mecanismo de auto adaptación para la generación de los patrones favorables (o sea, los buenos bloques constructores) de la cruza. Otros tipos de cruza existentes dependen del tipo de representación que se esté utilizando, como por ejemplo, la cruza para permutaciones, ya que en este tipo de problemas no debe haber valores repetidos, por lo tanto, se necesita un tipo de cruza especial; o cuando se utiliza representación con número reales, es preciso definir otro tipo de cruza que permita generar diversos números reales y así poder explorar mejor el espacio de búsqueda.

3.9. Métodos de Mutación

La mutación es considerada un operador secundario en los algoritmos genéticos y por tanto no se aplica demasiadas veces sobre los cromosomas como es el caso de la cruza. Sin embargo, es fundamental para asegurar que ningún alelo se pierda irremediablemente de la población durante la evolución del algoritmo.

Básicamente la mutación toma la información de un gen y cambiar el valor por alguno permitido, por ejemplo, en el caso de la representación binaria o con códigos de Gray, si se encuentra un 1 al mutar se cambiará el valor por un 0 y viceversa. En otros casos, como en la representación real, se puede elegir un número aleatorio que esté en el rango permitido e intercambiar el valor del gen.

Otro caso especial de mutación ocurre en aquellos algoritmos donde se utilicen permutaciones, por lo que será necesario aplicar una técnica con la que se puedan conservar las reglas de las mismas. Un ejemplo de esto es extraer un número existente en el cromosoma y luego seleccionar un lugar para anexarlo nuevamente, con esto se sigue conservando la restricción de que no contenga números repetidos.

Debido a que la mutación es un operador secundario, la probabilidad de que ésta sea aplicada, por lo general es muy baja. Los porcentajes de probabilidad suelen estar entre el rango de 0,001 y 0,01. Hay quienes recomiendan que la probabilidad de mutación sea de $\frac{1}{L}$, donde L es la longitud del cromosoma. También se ha sugerido comenzar con porcentajes de probabilidad e ir disminuyéndolos conforme pasa el número de generaciones para que al final se utilicen porcentajes de mutación pequeños. Esto con la finalidad de que al inicio se explore más el espacio de búsqueda y al final tenga una mejor convergencia.

3.10. Métodos de Reemplazo

Una vez que se produce la descendencia, un método debe determinar cuáles de los miembros actuales de la población deben permanecer en la próxima generación. Básicamente hay dos clases de métodos para el mantenimiento de la población [20]:

- Actualización generacional: El esquema básico de la actualización generacional consiste en producir N hijos de una población de tamaño N para formar la población del siguiente paso (generación), y esta nueva población reemplaza completamente a la generación de padres, implicando que un individuo puede reproducirse solamente con individuos de su misma generación.

- Actualización gradual: En este tipo de actualización los nuevos individuos se insertan en la población tan pronto como se crean, produciendo el reemplazo de otro miembro de la población. El individuo a eliminar puede seleccionarse como el peor miembro de la población (produciendo una presión selectiva muy alta), o como el miembro más antiguo de la población.

Como parte fundamental del algoritmo, debe considerarse una política de *Elitismo*, la cual consiste en nunca reemplazar a los mejores individuos en la población con soluciones inferiores, reteniendo siempre la mejor solución encontrada hasta el momento. Como desventaja, es importante tener en cuenta que el elitismo puede fomentar la convergencia prematura, haciéndole difícil a la solución escapar de un óptimo local.

3.11. Convergencia

Como es evidente, un algoritmo genético evoluciona su población de manera indefinida a menos que se le indique una condición de terminación. Esta condición de terminación puede ser:

- Un número fijo de generaciones.

- Cuando la mayoría o todos los miembros de la población sean similares o idénticos, es decir cuando la población haya convergido.

- Hasta que las mejoras en la población sean despreciables.

- O una combinación de las opciones antes expuestas.

Decir que una población ha *convergido* significa que ha perdido diversidad. Esta pérdida generalmente es ocasionada por la presión selectiva que tiende a reproducir a los mejores miembros de la población. Para contrarestar esto, sólo la mutación es capaz de agregar diversidad a la población, ya que la recombinación usada aisladamente sólo tiende a reorganizar el material genético y no a crear uno nuevo [20].

La *convergencia prematura* es uno de los principales problemas de los algoritmos genéticos. Este tipo de convergencia significa que la solución encontrada por el algoritmo es inaceptable. Una solución a esto puede ser incrementar la tasa de mutación o usar una tasa de mutación adaptativa, ya que permiten introducir nuevamente material genético perdido durante la evolución. Un tamaño de población mayor o una presión selectiva menor pueden también ayudar a evitar la convergencia prematura, aunque pueden

provocar un comportamiento aleatorio en el algoritmo. En contraposición a esto, si el algoritmo toma mucho tiempo para converger hacia la solución óptima se está frente al concepto de finalización lenta, que puede resolverse por medio de un incremento en la presión selectiva [20].

Como conclusión, solo el buen compromiso entre todos los parámetros del algoritmo conduce a un algoritmo óptimo capaz de encontrar buenas soluciones en un tiempo razonable.

Capítulo 4

Algoritmos Genéticos con Tamaño de Población Variable

El tamaño de la población es uno de los factores más importantes a tener en cuenta en cualquier uso de algoritmos genéticos y es crítico en muchas aplicaciones. Claramente, de este factor dependen la diversidad de la población y la presión selectiva.

En este capítulo se detallará una variante de los algoritmos genéticos tradicionales: los Algoritmos Genéticos con Tamaño de Población Variable (Genetic Algorithms with Varying Population Size - GAVaPS) presentados en [42]. Esta clase de algoritmos trata de resolver algunos de los principales problemas de los algoritmos genéticos permitiendo la autoadaptación del tamaño de la población en base al fitness de la misma.

4.1. Introducción

Como se expuso en el capítulo anterior, existen dos cuestiones muy importantes en el proceso evolutivo de búsqueda genética: la diversidad de la población y la presión selectiva. Claramente, ambos factores están influenciados por el tamaño de la población: si el tamaño de la población es pequeño, el algoritmo puede converger demasiado rápido, pero si el tamaño es muy grande, la performance del algoritmo suele verse degradada.

GAVaPS no usa ninguna de las variantes de selección vistas, sino que incorpora el concepto de *Edad* para un cromosoma. Este concepto depende del fitness del individuo y es equivalente al número de generaciones que el cromosoma ha "sobrevivido" hasta el momento (por lo que cambia en cada generación). Consecuentemente, el concepto de edad reemplaza al concepto de selección, ya que ahora todos tienen la misma probabilidad de reproducirse, y provoca que el tamaño de la población vaya variando a lo largo de las generaciones. Además de esto, otro concepto importante es introducido, el $TiempodeVida$, el cual indica el número de generaciones que el individuo puede sobrevivir.

En la figura 4.1 puede verse el pseudocódigo del algoritmo.

CrearPoblacion(*P*);
Evaluar el fitness y asignar tiempo de vida a P;
while (la población no ha convergido) do
    Incrementar la edad de cada individuo de P en 1;
    Seleccionar individuos para reproducción;
    Crear descendencia aplicando recombinación y/o mutación a los individuos seleccionados;
    Computar la aptitud y tiempo de vida de los nuevos individuos;
    Insertar la descendencia en P;
    Eliminar de P los individuos cuya edad sea mayor a su tiempo de vida;
end while
Salida : la mejor solución encontrada;

Figura 4.1: Algoritmo Genético con Tamaño de Población Variable

Analizando el código de la figura 4.1 puede observarse que este tipo de algoritmos presenta un crecimiento inicial significativo que comenzará a estabilizarse una vez que los individuos alcancen su tiempo de vida máximo. Por tal motivo, en estos algoritmos suele producirse una .[ex]plotación"inicial del espacio de soluciones para pasar, en una segunda etapa a un comportamiento más .[ex]plorador". El lograr la estabilidad esperada dependerá del tamaño de la población inicial, el tiempo de vida asignado a cada inidivuo y la estrategia de generación de nuevos individuos.

A continuación se analizarán las estrategias de asignación de tiempos de vida convencionales y luego se describirá la utilizada en el algoritmo propuesto en esta tesina.

4.2. Estrategias de asignación de tiempo de vida tradicionales

Como se explicó, GAVaPS no utiliza mecanismos de selección convencionales, sino que la presión selectiva la ejerce a través del concepto de tiempo de vida de cada individuo, por lo que la estrategia seleccionada para calcular el tiempo de vida debe considerar:

- Soportar individuos cuyo fitness sea mayor al fitness promedio.
- Ajustar el tamaño de la población actual de acuerdo al paso actual en la búsqueda, tratando de evitar el crecimiento exponencial.

A continuación se detallan las tres técnicas de asignación de tiempo de vida tradicionales.

Asignación Proporcional

El tiempo de vida del $Individuo_i$ es calculado como sigue:

$$TiempoDeVida(Individuo_i) = min(TV_{Min} + \frac{\eta * Fitness(Individuo_i)}{FIT_{Avg}}, TV_{Max}) \quad (4.1)$$

donde:

- TV_{Min} es el tiempo de vida mínimo que puede tomar $Individuo_i$.
- FIT_{Avg} es el promedio de todos los fitness de la población.
- TV_{Max} es el tiempo de vida máximo que puede tomar $Individuo_i$.
- $\eta = \frac{1}{2}(TV_{Max} - TV_{Min})$

Este tipo de asignación sigue la idea de la ruleta: el valor del tiempo de vida de un individuo es proporcional al valor de su fitness (dentro de los límites TV_{Min} y TV_{Max}). Esta aproximación tiene la falla de que solo utiliza el fitness promedio para caracterizar a la población actual. Entonces, para tener el tiempo de vida máximo, el individuo tiene que ser muy bueno (su fitness debe ser el doble que el fitness promedio), el cual significa que no todos los rangos de valores de tiempo de vida disponibles son correctamente usados.

Asignación Lineal

El tiempo de vida del $Individuo_i$ es calculado como sigue:

$$TiempoDeVida(Individuo_i) = TV_{Min} + 2 * \eta * \frac{Fitness(Individuo_i) - ABSFIT_{Min}}{ABSFIT_{Max} - ABSFIT_{Min}} \quad (4.2)$$

donde:

- $ABSFIT_{Min}$ es el valor absoluto del fitness mínimo de la población.
- $ABSFIT_{Max}$ es el valor absoluto del fitness máximo de la población.

Esta estrategia asigna el tiempo de vida proporcional a los valores máximo y mínimo de fitness encontrados en la población previa. Como consecuencia de la mejora del fitness de los individuos producida de una generación a otra, altos tiempos de vida son asignados en cada generación, causando un gran incremento del tamaño de la población.

Asignación Bilineal

El tiempo de vida del $Individuo_i$ es calculado como sigue:

$$TiempoDeVida(Individuo_i) = TV_{Min} + \eta * \frac{Fitness(Individuo_i) - FIT_{Min}}{FIT_{Avg} - FIT_{Min}} \quad (4.3)$$

si $FIT_{Avg} \geq Fitness(Individuo_i)$

$$TiempoDeVida(Individuo_i) = \frac{1}{2}(TV_{Min} + TV_{Max}) + \eta * \frac{Fitness(Individuo_i) - FIT_{Avg}}{FIT_{Max} - FIT_{Avg}} \tag{4.4}$$

si $FIT_{Avg} < Fitness(Individuo_i)$

donde:

- FIT_{Min} es el fitness mínimo de la problación.
- FIT_{Max} es el fitness máximo de la problación.

Este método trabaja con el promedio de los fitness dividiendo a la población en dos clases. El rango disponible para los tiempos de vida es separado en dos partes iguales y, dependiendo de la distancia del fitness de cada individuo al fitness promedio, el desplazamiento es determinado con el correspondiente segmento.

Segun las pruebas reportadas por Michalewicz en [42], la estrategia lineal es caracterizada por una mejor performance y un alto costo. Por otro lado, la estrategia Bilineal es menos costosa, pero la performance no es tan buena como en el caso lineal. Finalmente, la estrategia proporcional provee una performance media con un costo medio.

4.3. Estrategias de asignación de tiempo de vida por Clases

Basada en las observaciones de los métodos de asignación tradicionales, en [38] se detalla un método que usa una asignación de tiempo de vida con las siguientes características:

- Obtiene una distribución adecuada en el rango de valores permitidos, en el sentido de priorizar los individuos que tienen mejor fitness.
- Limita el crecimiento de la población para encontrar el óptimo.
- Evita la convergencia al óptimo local.
- Limita la dispersión de los individuos una vez que el área del óptimo haya sido localizada.

En esta estrategia, los individuos se agrupan en diferentes clases en función de su aptitud. Para esto, se selecciona la cantidad de clases a utilizar, lo que define las distintas agrupaciones. Para esto, se utiliza el algoritmo de clustering $k - medias$, detallado en [30, 32, 2] que define un "centro" o "media" para cada clase. Inicialmente, el algoritmo toma los primeros k valores de fitness diferentes como centros y conforma las clases con el resto de los individuos de la primera generación, midiendo su distribución en función de sus distancia más corta a un centro. Una vez que todos los individuos se distribuyen, los centros se calcularán de nuevo. El algoritmo sigue su iteraciones hasta que no haya cambios significativos en la centros de todas las clases.

Para cada nueva generación, el tiempo de vida de cada individuo está determinado en base a la agrupación de los individuos de la generación anterior. Dependiendo de la clase a la que pertenecen, y a su propio valor de fitness, los individuos recibirán su propio tiempo de vida, ya que cada clase tendrá un sub-rango asignado del total de tiempos de vida. Las clases varían desde las que contienen muy malos individuos (bajo fitness) a los que contienen muy buenos (alto fitness).

A continuación se detallan dos estrategias para la asignación de tiempos de vida por clase.

4.3.1. Asignación de Tiempo de Vida Fijo por Clase

Se divide el máximo tiempo de vida a asignar por la cantidad de clases, k. Esto permite saber el rango de tiempo que le corresponde a cada clase. Dentro de una misma clase, sus individuos recibirán un tiempo de vida proporcional a la clase a la que pertenecen y a la cantidad de individuos que se encuentran en su misma clase, de la siguiente forma:

$$TVClaseActual := MAX\_LT/k$$
$$TVClasesAnt := (ClaseMasCercana - 1) * TVClaseActual$$

$$Desplazamiento = (fitness[i] - Clase[ClaseMasCercana].MinFit)/ \\ abs(Clase[ClaseMasCercana].MaxFit - Class[ClaseMasCercana].MinFit))$$

$$TiempoDeVida[i] := trunc(TVClasesAnt + TVClaseActual * Desplazamiento)$$

donde

- $TVClaseActual$ es el rango del tiempo de vida asignado a cada clase (dada por MAX_LT sobre el número de clases).
- $ClaseMasCercana$ es el número de clase a la que pertenece cada individuo.
- $TVClaseActual$ es el rango de tiempo de vida de la clase a la pertenece el individuo.
- $TVClasesAnt$ es el rango de tiempo de vida asignado a las clases anteriores a $ClaseMasCercana$.

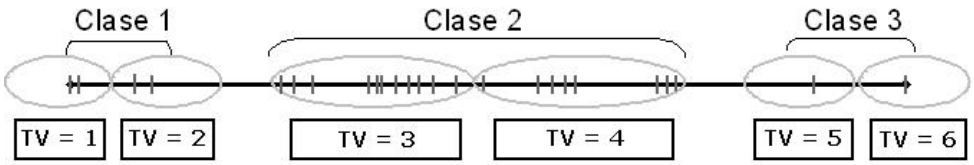


Figura 4.2: Ejemplo de Asignación de Tiempo de Vida Fijo por Clase

- $Clase[ClaseMasCercana].MinFit$ y $Clase[ClaseMasCercana].MaxFit$ son los valores de aptitud mínimo y máximo de la clase a la que pertenece el individuo en consideración.
- $Fitness[i]$ es el valor de aptitud del i-ésimo individuo de la población.

En la figura 4.2 se puede observar un ejemplo de este tipo de asignación con un tiempo de vida máximo de 6 iteraciones y una agrupación en 3 clases.

Nótese que el agrupamiento realizado con un método del estilo "winner take all" como es el caso del k-medias reúne en k grupos difrentes a los fitness de todos los individuos. La aplicación del cálculo de tiempo de vida fijo por clase no tiene en cuenta la cantidad de elementos de cada grupo sino que sólo utiliza los valores máximos y mínimos correspondientes a cada clase. De esta forma,los agrupamientos con muchos individuos sólo podrán ponder adecuadamente a sus miembros cuando el intervalo de fitness involucrado sea los suficientemente amplio.

4.3.2. Asignación de Tiempo de Vida Proporcional a la Cantidad de Individuos de cada Clase

A diferencia de la asignación anterior, ahora cada clase recibe un rango de tiempo de vida proporcional a la cantidad de elementos que contiene. Es decir, que los individuos pertenecientes a las clases numerosas podrán tener un rango de tiempo de vida más amplio, esto facilitará su discriminación. El cálculo es el siguiente:

$TotalAnt := 0$
for i :=1 to ClaseMasCercana - 1 do TotalAnt := TotalAnt + Clase[i].Cantidad
$TVAnterior := MAX\_LT * TotalAnt/TotalIndiv$
$TVClaseActual := MAX\_LT * Clase[ClaseMasCercana].Cantidad/TotalIndiv$

$$Desplazamiento = (fitness[i] - Clase[ClaseMasCercana].MinFit)/ abs(Clase[ClaseMasCercana].MaxFit - Clase[ClaseMasCercana].MinFit))$$

$$LifeTime[i] := trunc(TVAnterior + TVClaseActual * Desplazamiento)$$

donde

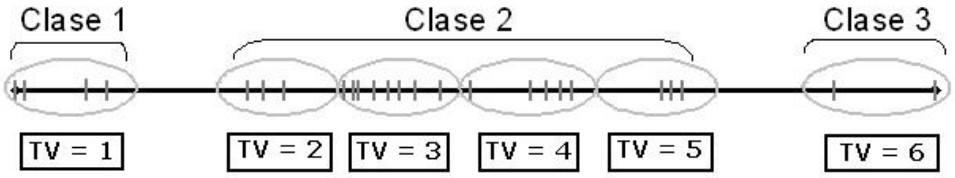


Figura 4.3: Ejemplo de Asignación de Tiempo de Vida Proporcional a la Cantidad de Individuos por Clase

- $Clases[ClaseMasCercana].Cantidad$ representa la cantidad toal de individuos de la clase más cercana.
- $TotalIndiv$ es la cantidad total de individuos de la población.

En la figura 4.3 se puede observar un ejemplo de este tipo de asignación con un tiempo de vida máximo de 6 iteraciones y una agrupación en 3 clases.

Estas dos formas de calcular los tiempos de vida de los individuos deben combinarse a fin de lograr una asignación correcta. En [38] se propone aplicar la asignación de tiempo de vida fijo durante un cierto porcentaje de la cantidad de generaciones máximas del algoritmo y en las restantes utilizar asignación de tiempo de vida variable. Esto se debe a que los agrupamientos iniciales se realizan sobre individuos que aun no están lo suficientemente adaptados y por consiguiente dan lugar a agrupamientos de tamaños muy disímiles. Si sobre estos agrupamientos se aplicara directamente la distribución de tiempos de vida fijo, muchos individuos recibirían tiempos de vida similares llevando al algoritmo a incrementar innecesariamente la cantidad de individuos de la población.

Como es sabido, el método $k-medias$ requiere que el valor de k se indique de antemano, pero este valor depende en gran medida del problema y del rango total de tiempos de vida. Por lo tanto, se debe tener en cuenta que el número de clases es directamente proporcional a la presión selectiva ejercida.

Capítulo 5

PSO con Tamaño de Población Variable

Hasta ahora, se han presentado dos métodos fundamentales dentro de lo que son las metaheurísticas de optimización: Optimización de Cúmulo de Partículas (en sus versiones *LBest* y *GBest*) y Algoritmos Genéticos (en sus versiones de Tamaño de Población Fijo y Variable).

En este capítulo se propone unir las ventajas de los dos métodos vistos, presentando una extensión original de PSO que incorpora los conceptos de edad y vecindad para permitir la variación del tamaño de la población.

5.1. Introducción

En el algoritmo PSO descripto, cada individuo permanece en continuo movimiento dentro del espacio de búsqueda y nunca muere. Este algoritmo necesita experiencia previa para la definición de la cantidad de soluciones que se van a manejar en cada iteración, de manera de no condicionar el resultado a obtener.

En esta tesina se propone generar una extensión original de PSO con tamaño de cúmulo variable con el objetivo de mejorar la relación de compromiso existente entre la velocidad de convergencia y la diversidad de la población. La variación de la población hace innecesario definir a priori el tamaño del cúmulo, evitando así el problema planteado anteriormente.

5.2. Cúmulos de Partículas con Tamaño de Población Variable

El algoritmo de Cúmulos de Partículas con Tamaño de Población Variable se basa en una modificación del proceso adaptativo que permite que las partículas se reproduzcan y mueran en función de su aptitud para resolver el problema planteado. Esto se realiza principalmente a través del concepto de edad que permite determinar el tiempo de permanencia de cada elemento dentro de la población. Además, dado que PSO tiende a

poblar rápidamente las zonas exploradas con buen fitness, para no poblar excesivamente un mismo lugar del espacio de soluciones, se analiza el entorno de cada individuo y se eliminan las peores soluciones de las zonas muy pobladas.

5.2.1. Tiempo de vida

Uno de los conceptos más importantes de la estrategia propuesta en este artículo es el tiempo de vida de una partícula ya que determina la duración de su permanencia dentro de la población. Dicho valor se expresa en cantidad de iteraciones, transcurridas las cuales, la partícula es eliminada. Este valor tiene una estrecha relación con la aptitud de la partícula y permite que los mejores individuos permanezcan en la población por mayor tiempo, influenciando el comportamiento del resto.

Para estimar el tiempo de vida de cada individuo de la población se utilizó el método de asignación por clases definido en [38] ya que ha demostrado poseer la capacidad de brindar buenos resultados con una cantidad de individuos muy inferior a la empleada por los métodos convencionales.

Como se detalla en el capítulo 4, en [38] los individuos de la población son agrupados según su valor de aptitud en k clases utilizando un método de clustering competitivo del tipo winner-take-all y el tiempo de vida es asignado de acuerdo al método fijo o proporcional por clase vistos en la sección 4.3.

5.2.2. Inserción de Partículas

La inserción de partículas tiene dos objetivos: incrementar la velocidad de convergencia incorporando individuos en las zonas menos pobladas y compensar la eliminación de partículas provocada por el cumplimiento de los respectivos tiempos de vida. Determinar los lugares convenientes, dentro del espacio de búsqueda, donde deben insertarse los nuevos individuos no es una tarea trivial. En realidad, se trata de una situación de compromiso entre la identificación de las zonas óptimas y la velocidad del proceso de inserción de los nuevos individuos.

En una primera aproximación, se planteó basar la inserción de nuevas partículas asignándole a las N ya existentes en el cúmulo una probabilidad de reproducción. Para calcular esta probabilidad, primero se analizó su entorno tomando un radio de distancia r calculado de la siguiente forma [49]:

$$d_i = min\{\|x_i - x_j\|; \forall j x_i, x_j \epsilon S; x_i \neq x_j\} \qquad i = 1..n \tag{5.1}$$

$$r = \frac{\sum_{i=1}^{n} d_i}{n} \tag{5.2}$$

Como puede observarse, r se calcula como el promedio de las distancias de cada partícula con su vecino más cercano y es el mismo para todas las partículas.

La probabilidad de reproducción de la $i-$ésima partícula se calculó de la siguiente forma [19]:

$$Prob_i = 1^{\smile}(CantVecinosExistentes_i/MAX\_CANTIDAD) \qquad i = 1..n \qquad (5.3)$$

siendo $CantVecinosExistentes$ la cantidad de partículas encontradas dentro del entorno del $i-$ésimo individuo de la población (definido en base al radio r) y $MAX\_CANTIDAD$ era un parámetro del algoritmo que representa la máxima cantidad de individuos permitida.

Cada individuo de la población, según su probabilidad de reproducción, podía generar un único descendiente aplicando mutación uniforme sobre la posición actual y copiando en el nuevo individuo su vector velocidad y su conocimiento social.

Luego de varias pruebas con esta forma de reproducción, se pudo observar que no permitía acotar el crecimiento de la población. Esto se debió a que, al concentrarse los individuos sobre la misma sección del espacio de búsqueda, el radio r decrecía rápidamente, dando demasiadas posibilidades de reproducción para las partículas del cúmulo, disparando así un crecimiento desmedido.

En vista de solucionar esto, se adoptó otra estrategia que divide el problema original en dos partes: en primer lugar busca determinar cuántas partículas es necesario incorporar para luego establecer dónde deben posicionarse dentro del espacio de búsqueda.

La cantidad de partículas incorporadas en cada iteración coincide con la cantidad de individuos aislados. Se considera un individuo aislado a aquel que no posea ningún vecino dentro de un radio r preestablecido (ecuaciones 5.1 y 5.2).

Solo resta determinar dónde posicionar estos nuevos individuos. El criterio adoptado fue el siguiente: el 20 % de estas nuevas partículas reciben el vector posición de los mejores individuos de la población pero su vector velocidad es aleatorio; el 80 % restante es random.

De esta forma, una parte de los nuevos individuos comenzarán a moverse desde las posiciones que mejor desempeño han demostrado hasta el momento pero con dirección y velocidad distintas a las de los mejores individuos. El 80 % restante permitirá llegar a explorar otras zonas del espacio de búsqueda.

Es importante remarcar que, la eficacia de la medida de distancia utilizada en 5.1 dependerá de la representación del espacio de búsqueda seleccionada. En caso de ser necesario, pueden verse otras alternativas en [17].

5.3. Algoritmo Propuesto

El algoritmo comienza con una población de N individuos generados al azar dentro del espacio de búsqueda y calcula para cada uno de ellos su fitness y tiempo de vida correspondientes.

Durante el proceso, los individuos se desplazan según las ecuaciones 2.1 y 2.2. La inercia utilizada para actualizar los vectores de velocidad es ajustada de acuerdo a la ecuación 2.3.

El uso de un peso de inercia variable facilita la adaptación de la población. Un valor de w alto al comienzo de la evolución le permite a las partículas realizar movimientos grandes ubicándose en distintas posiciones del espacio de búsqueda. A medida que avanza el número de iteraciones, el valor de w se reduce permitiéndoles realizar un ajuste más fino.

A partir de las nuevas posiciones dentro del espacio de búsqueda, se recalcula el valor de fitness de los individuos y se obtiene el radio r según la ecuación 5.2.

Luego, se crean tantos individuos nuevos como partículas existan en la población sin vecinos dentro de este radio. Estos nuevos individuos tendrán vectores de velocidad random, dentro de los rangos permitidos. El 20 % de estas nuevas partículas recibirán los vectores de posición de los mejores individuos de la población y el 80 % restante tendrán vectores de posición random. Para estas nuevas partículas, se evalúa su fitness y se las incorpora a la población. Utilizando la población completa se calcula el tiempo de vida de los recientemente incorporados.

Se decrementa en 1 el tiempo de vida de todos los individuos y aquellos que hayan alcanzado el valor cero son eliminados de la población.

El algoritmo propuesto utiliza elitismo por lo que el mejor individuo de cada iteración es preservado. De esta forma se garantiza que la población tendrá al menos una partícula. Esto se realiza reemplazando a la partícula con menor fitness por la mejor de la iteración anterior.

Finalmente, el algoritmo termina cuando se cumple una de las siguientes condiciones:

- Se alcanzó la cantidad máximas de iteraciones indicadas inicialmente.
- El mejor fitness no se ha modificado durante el 15 % de las iteraciones totales.

La figura 5.1 contiene el pseudocódigo del algoritmo descripto.

Pop= $CrearPoblacion(N)$;
CalcularFitness(Pop);
CalcularTiempoDeVida(Pop,2);
w←INERCIAMAXIMA;
while (no se alcance la condición de terminación) do
    for i = 1 to size(Pop) do
        Evaluar Partícula x_i del Cúmulo Pop;
        if fitness(x_i) es mejor que el $fitness(pBest_i)$ then
            $pBest_i \leftarrow x_i$; $fitness(pBest_i) \leftarrow fitness(x_i)$;
        end if
    end for
    Salvar la partícula con el fitness máximo;
    for i = 1 to size(Pop) do
        Marcar g_i de acuerdo al criterio de vecindario usado;
        $v_i \leftarrow w.v_i + (\varphi_1.rand_1.(pBest_i - x_i) + \varphi_2.rand_2.(g_i - x_i)$;
        $x_i \leftarrow x_i + v_i$;
    end for
    CalcularFitness(Pop);
    CalcularRadio(Pop, CantHijos, Sentenciados);
    Nuevos = CrearPoblacion(CantHijos);
    Asignar 20 %de estos nuevos individuos los vectores;
    de posición de los mejores individuos de Pop;
    CalcularFitness(Nuevos);
    VerEntorno(Pop, Sentenciados, CantHijos);
    Pop = Pop ∪ Nuevos;
    CalcularTiempoDeVida(Pop);
    if (Iteración Actual es mayor que 5 %de las ITERACIONESTOTALES))
            CalcularTiempoDeVida(Pop, 1);
        else CalcularTiempoDeVida(Pop, 2);
    end
    Decrementar en 1 el tiempo de vida de cada partícula;
    Remover las partículas con tiempo de vida nulo;
    Reemplazar el peor individuo por el salvado al comienzo de la iteración;
    w ← Modificar dinámicamente la inercia;
end while
Salida : la mejor solución encontrada;

Figura 5.1: Algoritmo del método VarPSO propuesto

La función *CrearPoblacion* recibe como parámetro la cantidad de partículas a crear y devuelve un cúmulo con vectores de posición y velocidad random dentro de los límites establecidos y con tiempos de vida nulos. Para estimarlos es preciso evaluar previamente el fitness de cada individuo.

El proceso $CalcularTiemposDeVida$ recibe un cúmulo completo y sólo calcula el tiempo de vida correspondiente a las partículas que poseen tiempo de vida nulo al momento de hacer la invocación. No es posible aplicarlo únicamente al cúmulo $Nuevos$ porque el cálculo depende del agrupamiento de todos los individuos según su valor de fitness. El segundo parámetro corresponde al tipo de cálculo que debe realizar y vale 1 para la asignación de tiempo de vida fijo y 2 para la asignación de tiempo de vida variable.

El cálculo del radio según la ecuación 5.2 se realiza dentro del proceso $CalcularRadio$ que recibe como parámetro el cúmulo completo y devuelve la cantidad de partículas nuevas que deben insertarse en la población. Este mismo módulo es el encargado de evitar la concentración de varias partículas en un mismo lugar del espacio de búsqueda, por tal motivo, también retorna la lista de individuos que tienen vecinos muy próximos. Dichas partículas son eliminadas en el módulo $VerEntorno$ en función de su fitness y la cantidad de hijos generados.

Capítulo 6

Problema Abordado

La extensión original de PSO con Tamaño de Cúmulo Variable que se presenta en esta tesina, y que se ha denominado VarPSO, busca mejorar la relación de compromiso existente entre la velocidad de convergencia y la diversidad de la población.

En este capítulo se propone mostrar los resultados obtenidos por VarPSO en la resolución de funciones matemáticas complejas comparándolo con la versión PSO original y los Algoritmos Genéticos de población fija y variable.

6.1. Funciones Matemáticas Complejas

El algoritmo propuesto en este trabajo fue utilizado para obtener el valor mínimo de distintas funciones. Por lo tanto, cada partícula contiene en su vector posición los valores de los argumentos de la función. La aptitud de cada partícula o individuo se calcula de la siguiente forma:

$$(c\_max - Valor\_de\_la\_Particula) \tag{6.1}$$

donde c_max representa una cota superior de la función en el intervalo a optimizar y Valor_de_la_Partícula es el resultado de evaluar la función en el vector posición de la partícula correspondiente.

A continuación se detallan las funciones utilizadas. Para cada una de ellas se indica el intervalo utilizado para determinar el espacio de búsqueda y el valor de c_max empleado para calcular el fitness.

$F1(x,y) = x^2 + y^2 \qquad x, y \epsilon [-1,5]; c\_max = 50$

$F2(x) = -x * sin(10 * \pi * x) + 1 \qquad x \epsilon [-2,1]; c\_max = 3$

$F3(x,y) = 0{,}5 + \frac{(sin(\sqrt{x^2+y^2+4}))^2 - 0{,}5}{(1+0{,}001.(x^2+y^2))^2} \qquad x, y \epsilon [-50,50]; c\_max = 1$

$$F4(x_1, x_2) = \frac{1}{0{,}002+\sum_{j=1}^{25}\frac{1}{50j+\sum_{i=1}^{2}(x_i-a_{ij})^6}} \quad x_1, x_2 \epsilon [-50, 50]; c\_max = 500$$

$$a_{ij} = \begin{pmatrix} -32 & -16 & 0 & 16 & 32 & -32 & -16 & 0 & 16 & 32 & -32 & -16 & 0 & 16 & 32 & -32 & -16 & 0 & 16 & 32 & -32 & -16 & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & -16 & -16 & -16 & -16 & 0 & 0 & 0 & 0 & 0 & 16 & 16 & 16 & 16 & 16 & 32 & 32 & 32 & 32 & 32 \end{pmatrix}$$

$$F5(x_1, x_2) = \frac{1}{0{,}002+\frac{1}{1+(x_1-1)^{12}+(y_1+1)^{12}}}) \quad x_1, x_2 \epsilon [-200, 200]; c\_max = 500$$

$$F6(x_1, x_2) = \frac{1}{0{,}002+\sum_{j=1}^{3}\frac{1}{50j^2+\sum_{i=1}^{2}(x_i-b_{ij})^{12}}} \quad x_1, x_2 \epsilon [-50, 50]; c\_max = 500$$

$$b_{ij} = \begin{pmatrix} -30 & 16 & 30 \\ -40 & -32 & 35 \end{pmatrix}$$

Las seis funciones presentan un único mínimo. Cada una de ellas busca medir un aspecto diferente del algoritmo propuesto. $F1$ permite analizar la precisión de la solución obtenida. $F2$, $F3$ y $F4$ muestran su capacidad para moverse sobre un espacio de búsqueda con valores de fitness muy cambiantes. Nótese que la función $F4$ es una modificación de la función 5 de De Jong. Las funciones $F5$ y $F6$ fueron introducidas para analizar la capacidad exploratoria del algoritmo. En $F5$ aparece un único hueco que lleva al mínimo dentro de una superficie totalmente plana. $F6$ es similar pero presenta tres huecos con profundidades muy distintas.

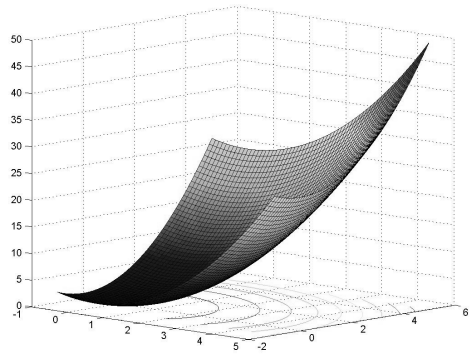
Los cuadros 6.1, 6.2, 6.3, 6.4, 6.5 y 6.6 muestran las gráficas de las funciones utilizadas.

6.2. Resultados obtenidos

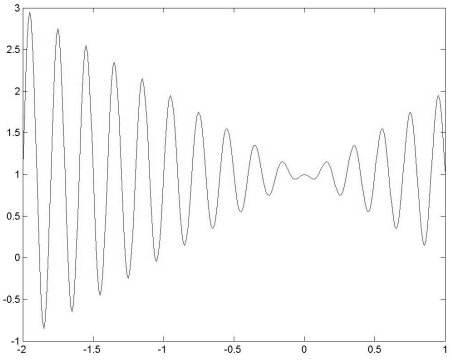
Para cada función se realizaron 100 pruebas utilizando en cada caso una cantidad máxima de 500 iteraciones.

Para el caso de PSO, se utilizó la versión con codificación continua. Los valores de aprendizaje cognitivo y social utilizados, φ_1 y φ_2 descriptos en la ecuación 2.2, fueron ambos en 0,5. Los valores de inercia entre 0,2 y 1,5. El rango de velocidades permitidas fue establecido entre $-0{,}5$ y 0,5.

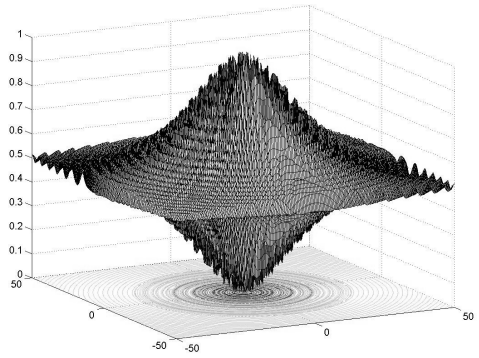
La cantidad de clases utilizadas para calcular el tiempo de vida de los individuos fue 4. Se realizaron pruebas con valores entre 2 y 10 confirmando que un valor alto para el número de clases, mejora la distinción del fitness entre los individuos pero incrementa sensiblemente el tamaño de la población. Por el otro lado, si la cantidad de clases es muy baja, el tamaño del cúmulo puede disminuir considerablemente. Un valor de 4 clases resulta adecuado para la minimización de las funciones antes indicadas.



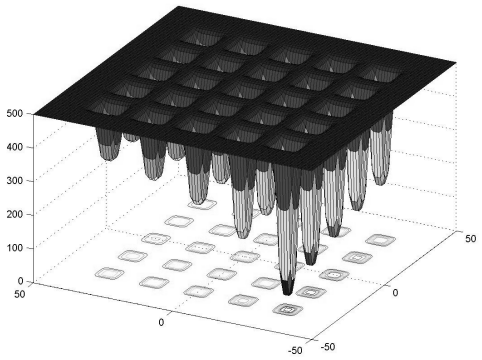
Cuadro 6.1: Gráfico de la función F1



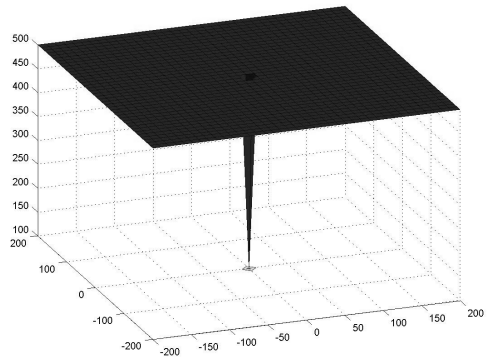
Cuadro 6.4: Gráfico de la función F2



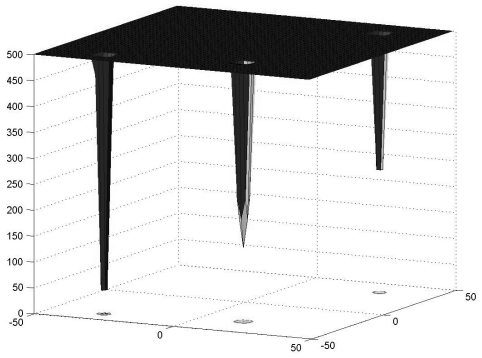
Cuadro 6.2: Gráfico de la función F3



Cuadro 6.5: Gráfico de la función F4



Cuadro 6.3: Gráfico de la función F5



Cuadro 6.6: Gráfico de la función F6

El valor utilizado como tiempo de vida máximo fue de 9 iteraciones salvo en las funciones $F5$ y $F6$ que se utilizó un valor de 12.

Para el caso de los Algoritmos Genéticos, se utilizó una representación binaria basada en códigos gray descripta en la sección 3.4 con una longitud del gen de 8 bits. Para representar números reales, ya que era necesario mantener las variables involucradas entre un rango especificado previamente en el función, se utilizó la siguiente transformación:

$$Nro = LI + \frac{ValorDeCadenaBinaria}{(2^{LongGen} - 1)}(LS - LI) \tag{6.2}$$

Como método de selección se ha utilizado el método de la ruleta, junto a una cruza de un solo punto y mutación simple. Los valores correspondientes a la probabilidad de mutación y probabilidad de cruza fueron de 0,025 y 0,5, respectivamente.

Para la versión con Población Variable, el método de asignación de tiempo de vida y reproducción utilizados fueron los mismos que para PSO con Población Variable. Se crean los hijos de la misma manera que la versión del algoritmo de población fija y se seleccionan, según el número de hijos necesarios en esa iteración, los que tienen los mejores fitness, el resto es descartado.

El figura 6.1 permite comparar los resultados obtenidos al aplicar el algoritmo propuesto en este artículo con el algoritmo PSO de población fija, AG y GAVaPS. Se realizaron pruebas con tamaño de población inicial 5, 10, 20, 30, 40, 50, 60 y 70 partículas. En todos los casos, los valores corresponden a los promedios de las 100 pruebas realizadas para cada función tomando el tamaño de población para la cual se obtuvo el mejor fitness máximo promedio.

Como puede observarse, en todos los casos el método propuesto es superior o igual a la solución basada en población fija y superior a ambas versiones de Algoritmos Genéticos, presentando una mayor diversidad de población. Además, salvo la función $F1$ para la cual es simple llegar al fitness máximo, el método propuesto utiliza una población inicial inferior a los métodos de población fija. Si se analiza la cantidad de partículas desplazadas en promedio en cada caso puede verse que, en la mayoría de las funciones, el método propuesto realiza menos de la mitad de trabajo que las soluciones con población fija. Esto último no se verifica para las funciones $F1$ y $F6$. En el caso de $F1$, se debe a la simplicidad de la función que contrasta con el comportamiento explorador del método propuesto y en el caso de $F6$, se debe a que el método no converge prematuramente como su par de población fija sino que continua analizando el espacio de búsqueda obteniendo mejores soluciones.

La figura 6.2 muestra la capacidad del método propuesto para adaptarse a la superficie de búsqueda hallando una solución casi óptima independiente del tamaño de la población inicial. También puede observarse que los métodos de población fija ofrecen resultados

Función 1						
Método	# Iterac.Promedio	Fit.Minimo	Fit. Medio	Fit.Maximo	Pob.Inicial	Pob.Final
gBest PSO	106.9000	48.0544	49.2226	50.0000	20.00	20.00
lBest PSO	142.08	48.0373	49.2735	50	20.00	20.00
gBest VAR PSO	181.8900	30.4806	45.3319	49.9996	30.00	33.80
lBest VAR PSO	169.82	30.3254	45.0502	49.9995	30.00	32.80
GA	156.7800	30.0013	44.1076	49.5687	20.00	20.00
GAVaPS	193.43	30.2519	44.9702	49.7632	30.00	35.34

Función 2						
Método	# Iterac.Promedio	Fit.Minimo	Fit. Medio	Fit.Maximo	Pob.Inicial	Pob.Final
gBest PSO	102.6700	1.6791	2.1150	3.8466	60.00	60.00
lBest PSO	102.4	1.7325	2.0647	3.8456	70.00	70.00
gBest VAR PSO	115.2200	1.3443	2.3861	3.8489	40.00	13.10
lBest VAR PSO	122.18	1.2349	2.3705	3.8489	30.00	17.70
GA	108.6500	1.5672	2.0176	3.8376	60.00	60.00
GAVaPS	125.87	1.5601	2.2719	3.8412	40.00	26.98

Función 3						
Método	# Iterac.Promedio	Fit.Minimo	Fit. Medio	Fit.Maximo	Pob.Inicial	Pob.Final
gBest PSO	231.0600	0.7288	0.9074	0.9762	70.00	70.00
lBest PSO	197.59	0.613	0.8607	0.9717	60.00	60.00
gBest VAR PSO	175.1700	0.3032	0.5857	0.9942	20.00	53.30
lBest VAR PSO	154.88	0.3025	0.5866	0.9936	40.00	62.60
GA	241.6500	0.4561	0.7165	0.9702	50.00	50.00
GAVaPS	160.43	0.5001	0.7989	0.9801	30.00	37.98

Función 4						
Método	# Iterac.Promedio	Fit.Minimo	Fit. Medio	Fit.Maximo	Pob.Inicial	Pob.Final
gBest PSO	284.1000	441.7729	445.3109	446.6185	60.0000	60.0000
lBest PSO	218.77	437.8757	443.5489	444.9094	70.00	70.00
gBest VAR PSO	132.0400	145.0876	285.1007	453.0275	30.00	30.70
lBest VAR PSO	124.64	156.9515	299.9142	454.5333	40.00	55.00
GA	308.6000	137.9200	301.7600	446.5891	60.00	60.00
GAVaPS	136.06	145.7615	317.54	446.621	40.00	43.00

Función 5						
Método	# Iterac.Promedio	Fit.Minimo	Fit. Medio	Fit.Maximo	Pob.Inicial	Pob.Final
gBest PSO	140.1900	468.3641	482.4536	484.0312	70.00	70.00
lBest PSO	138.37	448.4032	477.8286	479.0419	70.00	70.00
gBest VAR PSO	117.0100	171.4216	280.7481	494.0287	40.00	16.20
lBest VAR PSO	121.72	134.1485	238.5021	496.0285	30.00	23.80
GA	143.2000	178.8501	298.3095	482.8932	60.00	60.00
GAVaPS	125.8	180.6933	305.7853	484.1792	30.00	28.90

Función 6						
Método	# Iterac.Promedio	Fit.Minimo	Fit. Medio	Fit.Maximo	Pob.Inicial	Pob.Final
gBest PSO	103.2700	373.4571	391.0167	391.8489	60.00	60.00
lBest PSO	115.49	404.7819	440.9218	442.8571	50.00	50.00
gBest VAR PSO	92.5600	71.9364	194.3381	443.2076	40.00	85.70
lBest VAR PSO	108.44	120.5613	228.8209	443.411	20.00	52.90
GA	112.9000	250.7852	312.1945	390.8764	40.00	40.00
GAVaPS	110.4	110.6732	350.6721	408.6531	30.00	61.40

Figura 6.1: Resultados obtenidos con PSO y AG, tanto con población fija como con población variable

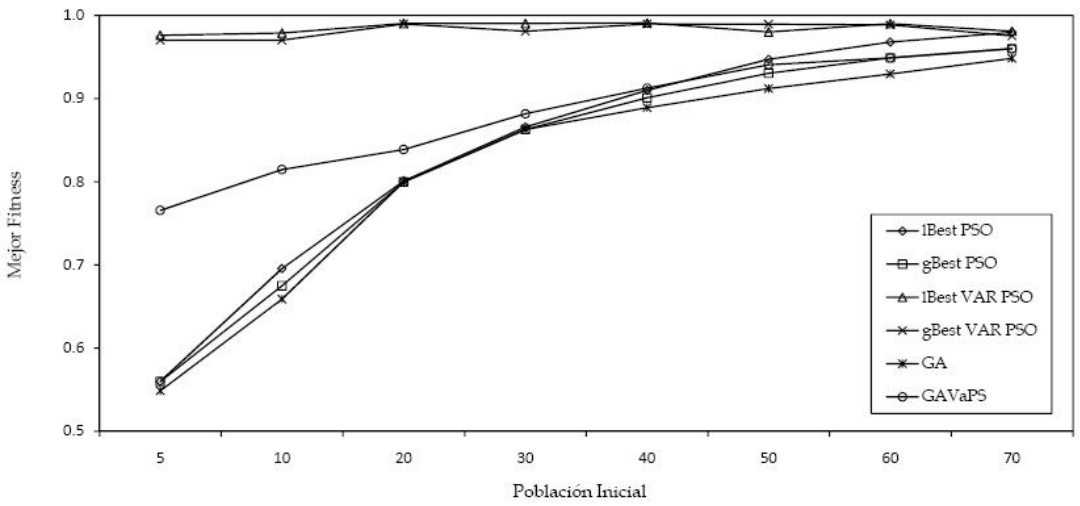


Figura 6.2: Fitness promedio máximo obtenido para distintos valores de población inicial. Cada punto es el resultado de promediar los mejores fitness de cada una de las 600 pruebas realizadas para cada método utilizando una población inicial determinada.

correctos cuando se utiliza la población inicial adecuada. Cada punto de la figura 6.2 corresponde al fitness máximo promedio de las 600 pruebas (100 para cada función) realizadas para cada tamaño de población inicial. En cada caso, el fitness ha sido escalado linealmente a $[0, 1]$ dividiéndolo por el fitness máximo correspondiente a la función.

Si se analiza la cantidad de iteraciones promedio realizada por cada método en función del tamaño de la población inicial puede observarse que para las soluciones de población fija tiende a incrementarse levemente mientras que para las de población variable decrece rápidamente a medida que la población inicial aumenta. Esto se encuentra representado en la figura 6.3 y refleja el comportamiento de cada método. Los de población fija dependen sólo del desplazamiento de las partículas iniciales mientas que las versiones de población variable realizan un aumento inicial de la población que les permite explorar rápidamente un área más amplia del espacio de búsqueda llegando al óptimo con un número de iteraciones menor.

Finalmente, las figuras 6.4 y 6.5 muestran el crecimiento promedio de la población para las dos variantes del método propuesto. Como puede observarse, el comportamiento es muy similar en los dos casos y tiene una fase de crecimiento, dentro de las primeras 30 iteraciones, seguida de una fase de reducción y estabilización.

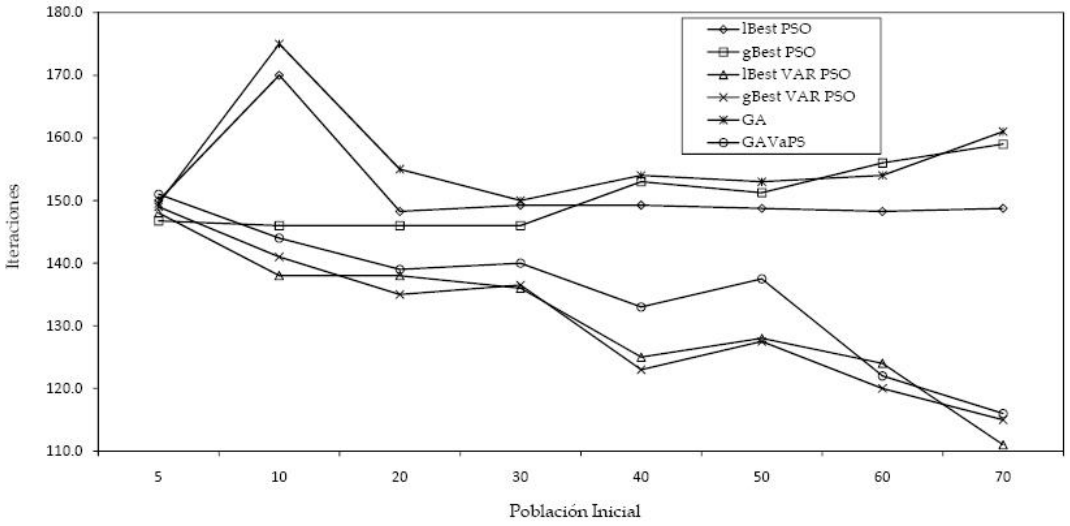


Figura 6.3: Variación del número de iteraciones promedio necesarias para obtener el mejor fitness en función del tamaño de la población inicial.

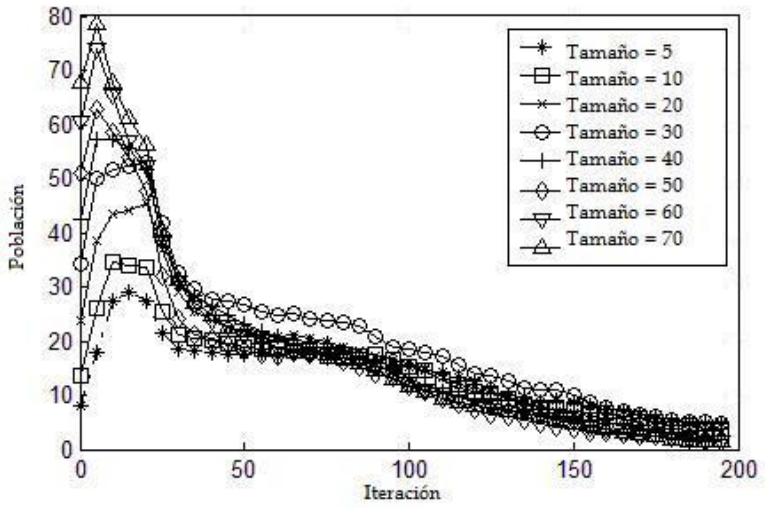


Figura 6.4: Tamaño promedio de la población utilizando gBest VAR PSO. Los valores corresponden al promedio de las primeras 200 iteraciones de las 600 pruebas realizadas (100 sobre cada función) con este método.

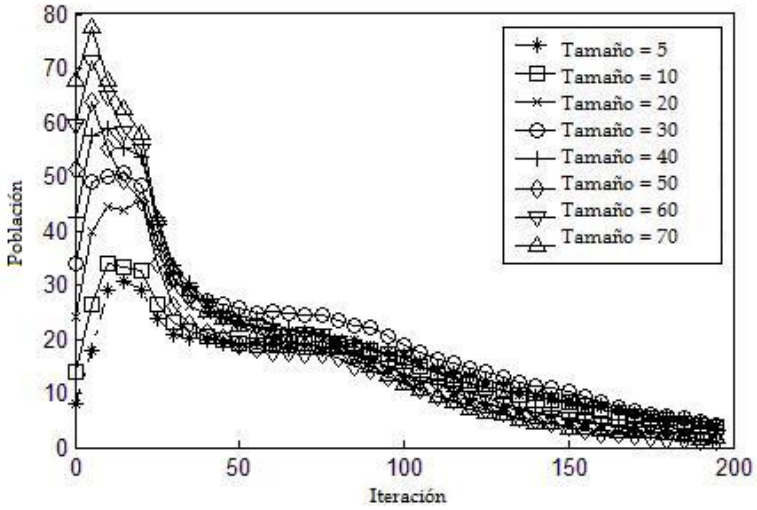


Figura 6.5: Tamaño promedio de la población utilizando lBest VAR PSO. Los valores corresponden al promedio de las primeras 200 iteraciones de las 600 pruebas realizadas (100 sobre cada función) con este método.

Capítulo 7

Conclusiones y trabajos futuros

Durante el transcurso de esta tesis, se ha profundizado en las metaheurísticas de optimización, particularmente dentro de los Algoritmos genéticos y Algoritmos basados en Optimización de Cúmulos de Partículas. Se han estudiado todos los mecanismos que estos poseen para desarrollar la evolución de las soluciones, junto con las ventajas de poseer o no el tamaño de la población variable durante su ejecución.

En vistas de que no se conocía una técnica eficiente de variabilidad de la población para PSO, se ha ideado una estrategia basada en cúmulos de partículas (PSO) con tamaño de población variable basado en los conceptos de edad y vecindario. De esta forma, no es necesario definir a priori la cantidad de soluciones a utilizar, evitando así condicionar la calidad de la solución a obtener.

En los comienzos del diseño del algoritmo, se pensó basar la reproducción de partículas en una probabilidad cuya variación dependía de la cantidad de vecinos que tuviera el individuo al momento de reproducirse. La nueva partícula fue generada como una copia de su padre con una mutación uniforme en su posición. Luego de varias pruebas, este método demostró no ser efectivo, ya que se pudo observar que no permitía acotar el crecimiento de la población. Esto se debió a que los individuos se concentraban sobre la misma sección del espacio de búsqueda, haciendo que el radio r decreciera rápidamente, otorgando demasiadas posibilidades de reproducción para las partículas del cúmulo.

En vistas del problema anterior, se diseñó el método de inserción de partículas que divide el problema en dos partes: evalúa cuántas partículas es necesario incorporar y luego establece dónde deben posicionarse dentro del espacio de búsqueda. Esto no sólo permitió acotar la reproducción excesiva, sino que permitió que una pequeña parte del cúmulo intensificara las zonas de mejor fitness, mientras que el resto de los nuevos individuos exploraran nuevas zonas del espacio de búsqueda, otorgándole diversidad al cúmulo.

El otro concepto importante para la variabilidad del tamaño fue el de tiempo de vida basado en asignaciones por clases (tanto fija como proporcional). Este concepto permitió acotar el tamaño de la población brindándole un tiempo de vida bajo a aquellas soluciones

con fitness menor, y una tiempo de vida mayor a aquella soluciones buenas, otorgando al algoritmo la capacidad de ejercer una adecuada presión selectiva sobre los individuos.

El algoritmo diseñado, en sus versiones *gBest PSO* y *lBest PSO*, se aplicó en diferentes funciones complejas. Los resultados obtenidos muestran que la estrategia propuesta en esta tesina permite obtener resultados con un mejor valor de aptitud que las producidas por ambas versiones de PSO con tamaño de población fijo y de Algoritmos Genéticos, tanto de población fija como variable, utilizando una menor cantidad de iteraciones.

Como trabajo futuro, y en virtud de la gran capacidad de exploración demostrada por el método propuesto en esta tesina, se está analizando la posibilidad de aplicar este método a la adaptación de redes neuronales. En este caso, cada partícula representará una red neuronal feedforward completa con arquitectura fija y a lo largo de las iteraciones se buscará adaptar los pesos de conexión. Como complejidad de esta aplicación que se presenta en esta tesis, se puede señalar a la medida de similitud que permitirá cuantificar adecuadamente la proximidad entre dos individuos del espacio de búsqueda.

Apéndice A

Sobre la Programación

A.1. Código para los algoritmos PSO y VarPSO

```
%=== PSO

clear
clc
%==== PARAMETROS DEL ALGORITMO =====
Salvar_Datos = 1;
Dibujo_onLine = 0;

POPSIZE_MINIMO = 1;

elitismo = 1;  % el mejor NO se mueve

MAX_CORRIDAS = 100; %== cantidad de veces que se ejecutara el mismo algoritmo ==
MAX_ITERA = 500; %== cant.maxima de iteraciones por cada corrida ==
MAX_ITERA_TVProp = 20; %== las primeras 20 iteraciones se asignara los TV de
                       %== manera proporcional
PROPORCION_MEJORES = 0.2; %== indica la proporcion del cumulo HIJOS que recibe
                          %== el vector posicion de los mejores actuales ==

%== Factor de inercia inicial y final
W_INI = 1.5;
W_FIN = 0.2;

alfa1 = 0.5;
alfa2 = 0.5;

%%== La 1er. fila de Ejecuciones tiene los valores para Usar_Poblacion_variable
% == la 2da. fila tiene los valores para Usar_gBest
% == la 3er. fila es el numeor de funcion a utilizar
% == la 4ta. fila es el tamanio de la poblacion inicial

Ejecuciones = [0  0  0  0  0  0  0  0  0  0  0  0  ;
               0  0  0  0  0  0  1  1  1  1  1  1  ;
               1  2  3  4  5  6  1  2  3  4  5  6  ;
               5  5  5  5  5  5  5  5  5  5  5  5  ];

nro_pruebas = size(Ejecuciones,2);
Usar_IgualItera = max(1,fix(0.15*MAX_ITERA));

for prueba = 1: nro_pruebas
  Usar_Poblacion_variable = Ejecuciones(1,prueba);
  Usar_gBest = Ejecuciones(2,prueba);
  popsizeIni = Ejecuciones(4,prueba);

  if (Usar_gBest == 1),
```

```
  nomMetodo = 'gBest';
else
  nomMetodo = 'lBest';
end

if ( Usar_Poblacion_variable == 1 ),
  nomMetodo = sprintf('%s%s',nomMetodo, 'VAR_PSO');
else
  nomMetodo = sprintf('%s%s',nomMetodo, 'FIX_PSO');
end

nro = Ejecuciones(3,prueba);

[cmax, limites, paso, NroFunc, x,y,z] = FuncionElegida(nro);

longIndiv = size(limites,2); %cant.de argumentos de la funcion

% rango de velocidad permitida para cada argumento de la funcion
% es una matriz de 2 filas y tantas columnas como argumentos tenga la funcion
% las columnas podrian ser distintas (distintos limites para cada argumento)
limVELOC =[ -0.5; 0.5] * ones(1,longIndiv);

estadist= ones(MAX_CORRIDAS, 6+longIndiv) .* -1;
Pobladores = ones(MAX_ITERA, MAX_CORRIDAS) .* -1;
LosMejoresFitness = ones(MAX_ITERA, MAX_CORRIDAS) .* -1;
LosPeoresFitness = ones(MAX_ITERA, MAX_CORRIDAS) .* -1;
LosFitnessPromedio = ones(MAX_ITERA, MAX_CORRIDAS) .* -1;

for corrida = 1:MAX_CORRIDAS
  %disp(sprintf('\n%s - corrida : %d',nomMetodo, corrida));
  Pop = CrearPoblacionInicialVariable(popsizeIni, limites, limVELOC);
  popsize = popsizeIni;

  if Dibujo_onLine==1,
    Dibujar(Pop, limites, paso, NroFunc,x,y,z);
  end

  itera=1;
  fitgBest = 0;

  igualMejor = 0;
  Pop = EvaluarFitness(cmax, NroFunc, Pop);

  Pobladores(itera, corrida) = popsize;
  LosMejoresFitness(itera, corrida) = max([Pop.fitness]);
  LosPeoresFitness(itera, corrida) = min([Pop.fitness]);
  LosFitnessPromedio(itera, corrida) = mean([Pop.fitness]);


  [Clases Pop] = AsignarTV(Pop,2);
  w_inercia = W_INI;

  while (itera < MAX_ITERA) & (igualMejor < Usar_IgualItera) &
        (popsize > 0) & (popsize < 300)

    for i=1:popsize
      if Pop(i).fitness > Pop(i).fitpBest
        Pop(i).pBest = Pop(i).individuo;
        Pop(i).fitpBest = Pop(i).fitness;
      end
    end

    % recordar quien es el mejor y donde esta por el elitismo
    Pop = EvaluarFitness(cmax, NroFunc, Pop);
    [FitMaxAnt, quienAnt] = max([Pop.fitness]);
    MejorIndividuo = Pop(quienAnt);

    if Usar_gBest == 1,
```

```
  gBest = Pop(quienAnt).individuo;
else
  medidas = Distancias(Pop);
  radio = mean(mean(medidas));
end

for i=1:popsize

  if Usar_gBest == 0,
    vecinos = find(medidas(:,i) <= radio);
    [FitMejorVecino, OrdenVecinos] = sort([Pop(vecinos).fitness]);
    MejorVecino = vecinos(OrdenVecinos(length(vecinos)));
    LBest = Pop(MejorVecino).individuo;

    Pop(i).velocidad = w_inercia * Pop(i).velocidad +
                       alfa1*rand*(Pop(i).pBest-Pop(i).individuo)
                       + alfa2*rand*(LBest-Pop(i).individuo);
  else
    Pop(i).velocidad = w_inercia * Pop(i).velocidad +
                       alfa1*rand*(Pop(i).pBest-Pop(i).individuo)
                       + alfa2*rand*(gBest-Pop(i).individuo);
  end

  for j=1:longIndiv
    if Pop(i).velocidad(j) < limVELOC(1,j),
      Pop(i).velocidad(j) = limVELOC(1,j); end
    if Pop(i).velocidad(j) > limVELOC(2,j),
      Pop(i).velocidad(j) = limVELOC(2,j); end
  end

  Pop(i).individuo = Pop(i).individuo + Pop(i).velocidad;
  %validar que el individuo no se vaya de los limites permitidos
  for j=1:longIndiv
    if Pop(i).individuo(j) < limites(1,j),
      Pop(i).individuo(j) = limites(1,j); end
    if Pop(i).individuo(j) > limites(2,j),
      Pop(i).individuo(j) = limites(2,j); end
    end
  end
  %== La poblacion se movio, se recalculan las aptitudes ==
  Pop = EvaluarFitness(cmax, NroFunc, Pop);

  %============ POBLACION VARIABLE ====================
  if (Usar_Poblacion_variable==1),
    %== nacimientos ==
    [CantHijos, sentenciados] = CalcularRadios(Pop);
    %== solos tiene los indices de los que se van a reproducir

    if CantHijos>0,  %== debe haber descendencia ==
      Hijos = CrearPoblacionInicialVariable(CantHijos, limites, limVELOC);
      [aptitudes orden] = sort([Pop.fitness]);
      popsize = length(Pop);
      if popsize<CantHijos
        [popsize CantHijos];
      end
      mini = max(1, popsize-floor(PROPORCION_MEJORES * CantHijos)+1);
      for h=popsize: -1:mini
        Hijos(popsize-h+1).individuo = Pop(orden(h)).individuo;
      end

      Hijos = EvaluarFitness(cmax, NroFunc, Hijos);

      Pop = VerEntorno(Pop, sentenciados, CantHijos);
      Pop = [Pop Hijos];

      if itera <= MAX_ITERA_TVProp,
        [Clases Pop] = AsignarTV(Pop,2);
      else
```

```
      [Clases Pop] = AsignarTV(Pop,1);
    end
  end

  %== Cumpleaños y muertes ==
  debenMorir = [];
  popsize = length(Pop);
  for i=1:popsize
    Pop(i).TV = Pop(i).TV - 1;
    if Pop(i).TV == 0,
      debenMorir = [debenMorir i];
    end
  end

  %== GARANTIZAR QUE LA POBLACION CUMPLE CON EL TAMAÑO MINIMO ==
  if popsize-length(debenMorir) < POPSIZE_MINIMO,
    % == van a quedar muy poquitos, salvemos a los mejores ===
    if popsize<=POPSIZE_MINIMO, %== no puede morir ninguno
      recuperados = debenMorir;
      debenMorir = [];
    else
      puedenMorir = popsize - POPSIZE_MINIMO;
      [aptitudes orden] = sort([Pop(debenMorir).fitness]);
      recuperados = debenMorir(puedenMorir+1:length(debenMorir));
      debenMorir = debenMorir(1:puedenMorir);
    end;
    for i=1:length(recuperados)
      Pop(recuperados(i)).TV = Pop(recuperados(i)).TVIni;
      for j=1:longIndiv
        Pop(recuperados(i)).velocidad(1,j) =
              rand*(limVELOC(2,j)-limVELOC(1,j))+limVELOC(1,j);
      end
    end
  end

  Pop(debenMorir) = [];
  popsize = length(Pop);
end    %== de if (Usar_Poblacion_variable==1)

%==================== FIN DE POBLACION VARIABLE ====================

%== verificar el elitismo ==
if (elitismo==1),
  [FitnessMax, OrdenDeAptitud] = sort([Pop.fitness]);
  quien = OrdenDeAptitud(popsize);
  FitMax = FitnessMax(popsize);

  %== el mejor (antes de mover a nadie) pasa seguro a la proxima
  %== generacion ==
   Pop(OrdenDeAptitud(1)) = MejorIndividuo;

   if (FitMax < FitMaxAnt),
      FitMax = FitMaxAnt;
      quien = OrdenDeAptitud(1);
   end
 end

 if Dibujo_onLine==1,
   Dibujar(Pop, limites, paso, NroFunc,x,y,z);
 end
 itera = itera + 1;


 [FitMax,quien] = max([Pop.fitness]);
 if FitMaxAnt == FitMax
   igualMejor = igualMejor + 1;
 else
   igualMejor = 1;
```

```
        end

        Pobladores(itera, corrida) = popsize;
        LosMejoresFitness(itera, corrida) = FitMax;
        LosPeoresFitness(itera, corrida) = min([Pop.fitness]);
        LosFitnessPromedio(itera, corrida) = mean([Pop.fitness]);
        w_inercia = W_INI - (W_INI-W_FIN) * (itera / MAX_ITERA);
      end

      disp(sprintf('Itera = %d, popsize = %d. Las ultimas %d tuvieron fitness = %f',
                     itera,popsize, igualMejor,FitMax));

      estadist(corrida,1) = corrida;
      estadist(corrida,2) = itera;
      estadist(corrida,3) = min([Pop.fitness]);
      estadist(corrida,4) = mean([Pop.fitness]);
      [maximo, quien] = max([Pop.fitness]);
      estadist(corrida,5) = maximo;
      estadist(corrida,6) = popsize;

      estadist(corrida,7:6+longIndiv) = Pop(quien).individuo;

  end

  if (Salvar_Datos == 1),
    disp(sprintf('Grabando las %d corridas de la Funcion %d',MAX_CORRIDAS,NroFunc));
    AVGPobladores = avg(Pobladores, 2);
    AVGLosMejoresFitness = avg(LosMejoresFitness, 2);
    AVGLosPeoresFitness = avg(LosPeoresFitness,2);
    AVGLosFitnessPromedio = avg(LosFitnessPromedio,2);

    if (Usar_gBest == 1),
       n = 'gBest';
    else
       n = 'lBest';
    end

    if ( Usar_Poblacion_variable == 1 ),
       n1 = 'VAR';
       auxi = sprintf('_%4.2f',PROPORCION_MEJORES);

       nom2 = sprintf('mediciones\\VAR_%s_Poblacion%d_%d_%4.2f',
                           n,NroFunc, popsizeIni,PROPORCION_MEJORES);
       save(nom2,'AVGPobladores','-ASCII', '-TABS');

    else
       n1 = 'FIX';
       auxi = '';
    end


    nom1 = sprintf('mediciones\\%s_%s_Fun%d_PopIni_%d%s',
                        n1,n,NroFunc,popsizeIni,auxi);
    save(nom1,'estadist','-ASCII', '-TABS');

    nom3 = sprintf('mediciones\\%s_%s_Fun%d_PopIni_%d_AVGLosMejores%s',
                       n1,n,NroFunc,popsizeIni,auxi);
    save(nom3,'AVGLosMejoresFitness','-ASCII', '-TABS');

    nom4 = sprintf('mediciones\\%s_%s_Fun%d_PopIni_%d_AVGLosPeores%s',
                       n1,n,NroFunc,popsizeIni,auxi);
    save(nom4,'AVGLosPeoresFitness','-ASCII', '-TABS');

    nom5 = sprintf('mediciones\\%s_%s_Fun%d_PopIni_%d_AVGLosMedios%s',
                        n1,n,NroFunc,popsizeIni,auxi);
    save(nom5,'AVGLosFitnessPromedio','-ASCII', '-TABS');

    %== solo para verificar ==
```

```
    nom6 = sprintf('mediciones\\detalles\\%s_%s_Fun%d_PopIni_%d_LosMejoresFitness%s',
                      n1,n,NroFunc,popsizeIni,auxi);
    save(nom6,'LosMejoresFitness','-ASCII', '-TABS');

    nom7 = sprintf('mediciones\\detalles\\%s_%s_Fun%d_PopIni_%d_LosPeoresFitness%s',
                      n1,n,NroFunc,popsizeIni,auxi);
    save(nom7,'LosPeoresFitness','-ASCII', '-TABS');

    nom8 = sprintf('mediciones\\detalles\\%s_%s_Fun%d_PopIni_%d_LosFitnessPromedio%s',
                     n1,n,NroFunc,popsizeIni,auxi);
    save(nom8,'LosFitnessPromedio','-ASCII', '-TABS');
  end
end



function [cmax, limites, paso, valorElegido, x,y,z] = FuncionElegida(nro);

valorElegido = nro;
x = [];
y = [];
z = [];
switch valorElegido
  case 1
    %===== f(x,y) = x^2 + y^2  ======
    cmax = 50; %Valor maximo utilizado para calcular el fitness
    limites = [-1 -1; 5 5];
    % c/columna representa el rango en el que varia cada argumento de la funcion
    paso = 0.1;   %Precision para el dibujo de la funcion
    [x,y] = meshgrid(limites(1,1):paso:limites(2,1), limites(1,2):paso:limites(2,2));
    z= x.^2 + y.^2;

  case 2
    %===== f(x) = sin(x * pi ...) ======
    cmax = 3; %Valor maximo utilizado para calcular el fitness
    limites = [-2; 1];
    % c/columna representa el rango en el que varia cada argumento de la funcion
    paso = 0.005;   %Precision para el dibujo de la funcion
    z = 0.5 + (sin(sqrt(x.^2 + y.^2)) .^ 2 - 0.5) ./ ((1 + 0.001 .* (x.^2 + y.^2)).^2);

  case 3
    %===== f(x,y) = 0.5+((sin(sqrt(x^2+y^2))^2-0.5)/(1+0.001*(x^2+y^2))^2) ======
    cmax = 1; %Valor maximo utilizado para calcular el fitness
    limites = [-50 -50; 50 50];
    % c/columna representa el rango en el que varia cada argumento de la funcion
    paso = 0.25;   %Precision para el dibujo de la funcion
    [x,y] = meshgrid(limites(1,1):paso:limites(2,1), limites(1,2):paso:limites(2,2));
    z = 0.5+((sin( sqrt(x.^2 + y.^2 + 4) ).^2-0.5)./(1+0.001*(x.^2 + y.^2)).^2);

  case 4
    %===== f(x,y) = 1/0.002 + sumatoria (50*j + (x(i,j)-a(1,k))^6 + (y(i,j)-a(2,k))^6)
    cmax = 500; %Valor maximo utilizado para calcular el fitness
    limites = [-50 -50; 50 50];
    % c/columna representa el rango en el que varia cada argumento de la funcion
    paso = 1;   %Precision para el dibujo de la funcion
    [x,y] = meshgrid(limites(1,1):paso:limites(2,1), limites(1,2):paso:limites(2,2));
    a = [ -32 -16 0 16 32 -32 -16 0 16 32 -32 -16 0 16 32 -32 -16 0 16 32 -32 -16 0 16 32;
          -32 -32 -32 -32 -32 -16 -16 -16 -16 -16 0 0 0 0 0 16 16 16 16 16 32 32 32 32 32];
    [fil,col] = size(x) ;
    for i=1:fil
      for j=1:col
        suma = 0;
        for k=1:25
          suma = suma + 1/(50*k + (x(i,j)-a(1,k))^6 + (y(i,j)-a(2,k))^6);
        end
        z(i,j)=1/(0.002+suma);
      end;
```

```
    end;

  case 5
    %===== f(x,y) = 1/(0.002+(1./(1+(x-1)^12+(y+1)^12))) ======
    cmax = 500; %Valor maximo utilizado para calcular el fitness
    limites = [-200 -200; 200 200];
    % c/columna representa el rango en el que varia cada argumento de la funcion
    paso = 2;   %Precision para el dibujo de la funcion
    [x,y] = meshgrid(limites(1,1):paso:limites(2,1), limites(1,2):paso:limites(2,2));
    z = 1./(0.002+(1./(1+(x-1).^12+(y+1).^12)));

  case 6
    %===== f(x,y) = 1/0.002 + sumatoria ((50*j)^2 + (x(i,j)-a(1,k))^12 + (y(i,j)-a(2,k))^12) ======
    cmax = 500; %Valor maximo utilizado para calcular el fitness
    limites = [-50 -50; 50 50];
    % c/columna representa el rango en el que varia cada argumento de la funcion
    paso = 1;   %Precision para el dibujo de la funcion
    [x,y] = meshgrid(limites(1,1):paso:limites(2,1), limites(1,2):paso:limites(2,2));
    a = [ -30  16  30; -40 -32 35];
    [fil,col] = size(x) ;
    for i=1:fil
      for j=1:col
        suma = 0;
        for k=1:3
          suma = suma + 1/(50*k^2 + (x(i,j)-a(1,k))^12 + (y(i,j)-a(2,k))^12);
        end
        z(i,j)=1/(0.002+suma);
      end;
    end;
end
```

```
function y = FUN(nro, x)
% x es una matriz con tantas columnas como puntos donde debe evaluarse la
% funcion y tantas filas como variables de entrada tenga la funcion
y = [];
long = length(x);
switch nro
  case 1
    puntosx = x([1:2:long]);
    puntosy = x([2:2:long]);
    y = puntosx.^2 + puntosy.^2;

  case 2
    y = ((-x) .* sin(10 * pi * x) )+ ones(1,length(x));

  case 3
    puntosx = x([1:2:long]);
    puntosy = x([2:2:long]);
    y = 0.5+((sin( sqrt(puntosx.^2+puntosy.^2 + 4) ).^2-0.5)./(1+0.001*(puntosx.^2+puntosy.^2)).^2);

  case 4
    puntosx = x([1:2:long]);
    puntosy = x([2:2:long]);
    a = [ -32 -16 0 16 32 -32 -16 0 16 32 -32 -16 0 16 32 -32 -16 0 16 32 -32 -16 0 16 32;
          -32 -32 -32 -32 -32 -16 -16 -16 -16 -16 0 0 0 0 0 16 16 16 16 16 32 32 32 32 32];
    long = length(puntosx);
    for i=1:long
      suma = 0;
      for k=1:25
        suma = suma + 1/(50*k+(puntosx(i)-a(1,k))^6+(puntosy(i)-a(2,k))^6);
      end
      y(i)=1/(0.002+suma);
    end;

  case 5
```

```
    puntosx = x([1:2:long]);
    puntosy = x([2:2:long]);
    y = 1./(0.002+(1./(1+(puntosx-1).^12+(puntosy+1).^12)));

  case 6
    puntosx = x([1:2:long]);
    puntosy = x([2:2:long]);
    a = [ -30  16  30; -40 -32 35];
    long = length(puntosx);
    for i=1:long
      suma = 0;
      for k=1:3
        suma = suma + 1/(50*k^2+(puntosx(i)-a(1,k))^12+(puntosy(i)-a(2,k))^12);
      end
      y(i)=1/(0.002+suma);
    end;

end




function p = CrearPoblacionInicialVariable(popsize, limites, limVELOC)
% genera un vector de popsize inidividuos aleatorios
% cada uno es una estructura con los valores necesarios

long = size(limites,2); %cantidad de argumentos de la funcion
p = [];
nuevo = [];  nuevo = zeros(1,long);
veloc = [];  veloc = zeros(1,long);

for i=1:popsize
  for j=1:long
    nuevo(1,j) = rand*(limites(2,j)-limites(1,j))+limites(1,j);
    veloc(1,j) = rand*(limVELOC(2,j)-limVELOC(1,j))+limVELOC(1,j);
  end
  p = [p struct('individuo', nuevo, 'velocidad', veloc, 'pBest', nuevo,
       'fitness',0,'fitpBest',0, 'TV',0, 'TVIni',0)];
end




function nada = Dibujar(Pop, limites, paso, NroFunc, x, y, z);

CantArgs = size(limites,2);

switch CantArgs
  case 1  % dibujo 2D
    hold off
    %Visualizar la funcion FUN en el intervalo indicado
    Intervalo = limites(1,1):paso:limites(2,1);

    y = FUN(NroFunc, Intervalo);
    plot(Intervalo,y);
    hold on
    % graficar los individuos

    Salida = FUN(NroFunc, [Pop.individuo]);
    plot([Pop.individuo],Salida,'*');
    pause(0.01);
  case 2 %dibujo 3D
    hold off
    mesh(x,y,z);

    hold on
    z=FUN(NroFunc, [Pop.individuo]);
    puntos = [Pop.individuo];
```

```
    long = length(puntos);
    puntosx = puntos([1:2:long]);
    puntosy = puntos([2:2:long]);

    plot3(puntosx, puntosy, z, '*');
    pause(0.01);
end



function Pop = EvaluarFitness(cmax, NroFunc, Pop)

% evaluar el fitness de todos los individuos
popsize = length(Pop);
Salida = FUN(NroFunc, [Pop.individuo]);
Fitness = abs(cmax * ones(1,popsize) - Salida);

for i=1:popsize
  Pop(i).fitness = Fitness(i);
end



function [Centros, Pop] = AsignarTV(Pop, tipo);
% TIPO vale 1 si se usa asignacion de TV fijo por clase y 2 si se usa
% asignacion de TV proporcional a la cant.de individuos por clase

CANT_CLASES = 4;
MAX_TV = 10;

popsize = length(Pop);
%atrib = length(Pop(1).individuo);

%=== Asignar los centros de las clases iniciales ===
% los primeros centros son random
mezcla= randperm(popsize);

Centros = zeros(1,CANT_CLASES);
CANT = min(CANT_CLASES, popsize);
for i= 1:CANT
  Centros(1,i) = Pop(mezcla(i)).fitness;
end


actual = zeros(1,popsize);
%== recalcular los centros hasta que se estabilicen
dife = 10;
while dife>1
  ant = actual;

  Sumas = zeros(1, CANT_CLASES);
  Cant = zeros(1,CANT_CLASES);
  for i=1:popsize
    clase = ClaseMasCercana(Pop(i).fitness, Centros);
    Sumas(1,clase) = Sumas(1,clase) + Pop(i).fitness;
    Cant(1,clase) = Cant(1,clase) + 1;
    actual(i) = clase;
  end

  %== recalcular los centros ==
  for k = 1:CANT_CLASES
     if Cant(k)>0,
        Centros(1,k) = Sumas(1,k) / Cant(k);
     end
  end
  dife = sum(abs(ant-actual));
```

```
end

%== ordenar los centroides en forma creciente por el fitness ==
[Centros orden] = sort(Centros);
Cant = Cant(orden);
actual = orden(actual);

%== 'Centros' contiene los centroides y 'Cant' la cant.de elementos en c/clase ==
%== 'actual' indica la clase de cada individuo ==
%== Calcular el fitness minimo y maximo por clase ==
FitMin = zeros(CANT_CLASES);
FitMax = zeros(CANT_CLASES);
for k = 1:CANT
  elems=[];
  elems = find(actual == k);
  if length(elems) > 0,
     FitMin(k) = min([Pop(elems).fitness]);
     FitMax(k) = max([Pop(elems).fitness]);
  end
end

% FALTA ASIGNAR LOS TV segun lo que diga TIPO.
cuales = [];
cuales = find([Pop.TV] == 0);

if tipo==1
  %==  TV fijo por clase ==
  AnchoClase = MAX_TV / CANT_CLASES;
  for i=1:length(cuales)
    c = actual(cuales(i));
    TVClasesAnt = (c - 1)*AnchoClase;

    if FitMax(c)== FitMin(c),
      despl = 1;
    else
      despl = (Pop(cuales(i)).fitness - FitMin( c ))/abs(FitMax(c)-FitMin(c));
    end

    Pop(cuales(i)).TV = max(1, floor(TVClasesAnt + AnchoClase * despl ));
    Pop(cuales(i)).TVIni = Pop(cuales(i)).TV;
  end
else
  %== TV proporcional a la cant.de individuos ==
  for i=1:length(cuales)
    c = actual(cuales(i));
    elemsAnt = sum(Cant(1:c-1));
    TVAnterior = MAX_TV * elemsAnt / popsize;
    TVClaseActual = MAX_TV * Cant(c) / popsize;
    if FitMax(c)== FitMin(c),
      despl = 1;
    else
      despl = (Pop(cuales(i)).fitness - FitMin( c )) / abs(FitMax(c)-FitMin(c));
    end

    Pop(cuales(i)).TV = max(1,floor(TVAnterior + TVClaseActual * despl ));
    Pop(cuales(i)).TVIni = Pop(cuales(i)).TV;
  end
end




function masCerca = ClaseMasCercana(quien, Centros);
%== calcula distancia euclidea entre 'quien' y c/u de las columnas de 'Centros'
[fil,col] = size(Centros);
distancias = sqrt(sum((Centros - quien*ones(1,col)).^2,1));
[menor,masCerca] = min(distancias);
```

```
function Pop = VerEntorno(Pop, sentenciados, CantNuevos);
%== sentenciados es un vector fila con los indices de los individuos muy proximos ==
%== tambien puede ser que sean el vecino del vecino del vecino ==
%== solo debe permanecer en la poblacion el mejor ==
sobrevivientes = 2;
if length(sentenciados) > sobrevivientes,
  CantBorrados = min( length(sentenciados)-sobrevivientes, CantNuevos);

  [ValoresFit, orden] = sort([Pop(sentenciados).fitness]);

  sentenciados(orden(CantBorrados+1: length(orden))) = [];
  %== solo quedan los que van a sobrevivir ==

  Pop(sentenciados) = [];
end
```

```
function d = Distancias(Pop)

popsize = length(Pop);

if popsize < 2,
  d = 0;
else
  d = zeros(popsize,popsize);
  P=[];
  for i = 1 : popsize
     P = [P [Pop(i).individuo]'];
  end

  for i = 1 : popsize-1
     p1 = P(:,1);
     P(:,1)=[];
     [fil,col] = size(P);

     distancias = sqrt(sum((P - p1*ones(1,col)).^2,1));
     d(i,i+1:popsize) = distancias;
     d(i+1:popsize, i) = distancias';
  end
end
```

```
function [CantHijos, sentenciados]= CalcularRadios(Pop)
% calcular la matriz de distancias entre los individuos de la poblacion

popsize = length(Pop);
if popsize < 3,
  CantHijos = 3;
  sentenciados = [];
else
  %== retirar del analisis el 10% de los mejores individuos ==

  d=Distancias(Pop); %= matriz de distancias ==
  masLejos = max(max(d));
  for i = 1 : popsize
    d(i,i)=masLejos;
  end
  [radiosMinimos, menores] = min(d);

  media = mean(radiosMinimos);
  desviacion = sum(abs(radiosMinimos - media))/popsize;
```

```
  CantHijos = length(find(radiosMinimos > media));

  sentenciados = find((radiosMinimos < (media-desviacion)));
end



function mean = avg(x, dim)

mean = sum(x .* (x>-1),2)./ (sum(x>-1,2) + (sum(x>-1,2) == 0));
```

A.2. Código para los algoritmos GA y GAVaPS

```
%=== GA


clear
clc
%==== PARAMETROS DEL ALGORITMO =====
Salvar_Datos = 1;
Dibujo_onLine = 0;
POPSIZE_MINIMO = 3;
elitismo = 1;  % el mejor NO se muere
longGen  = 8;        % longitud del genotipo
pCross  = 0.5;      % probabilidad de cruza
pMutation = 0.025;   % prob.abilidad de mutacion
MAX_CORRIDAS = 100; %== cantidad de veces que se ejecutara el mismo algoritmo ==
MAX_ITERA = 500; %== cant.maxima de iteraciones por cada corrida ==
MAX_ITERA_TVProp = 20; %== las primeras 20 iteraciones se asignara los TV de
                       % manera proporcional

%%== La 1er. fila de Ejecuciones tiene los valores para Usar_Poblacion_variable
% == la 2da. fila tiene los valores para Usar_gBest
% == la 3er. fila es el numeor de funcion a utilizar
% == la 4ta. fila es el tamanio de la poblacion inicial

Ejecuciones = [ 0  0  0  0  0  0  1  1  1  1  1  1 ;
                1  2  3  4  5  6  1  2  3  4  5  6 ;
                5  5  5  5  5  5  5  5  5  5  5  5 ];

nro_pruebas = size(Ejecuciones,2);
Usar_IgualItera = max(1,fix(0.15*MAX_ITERA));

for prueba = 1: nro_pruebas
    Usar_Poblacion_variable = Ejecuciones(1,prueba);
    popsizeIni = Ejecuciones(3,prueba);

    if ( Usar_Poblacion_variable == 1 ),
        nomMetodo = sprintf('%s%s', 'GAVaPS');
    else
        nomMetodo = sprintf('%s%s', 'GA');
    end

    nro = Ejecuciones(2,prueba);

    [cmax, limites, paso, NroFunc, x,y,z] = FuncionElegida(nro);

    longIndiv = size(limites,2); %cant.de argumentos de la funcion

    estadist= ones(MAX_CORRIDAS, 6+longIndiv) .* -1;
    Pobladores = ones(MAX_ITERA, MAX_CORRIDAS) .* -1;
    LosMejoresFitness = ones(MAX_ITERA, MAX_CORRIDAS) .* -1;
    LosPeoresFitness = ones(MAX_ITERA, MAX_CORRIDAS) .* -1;
    LosFitnessPromedio = ones(MAX_ITERA, MAX_CORRIDAS) .* -1;
```

```
for corrida = 1:MAX_CORRIDAS
    disp(sprintf('\n%s - corrida : %d',nomMetodo, corrida));
    Pop = CrearPoblacionInicialVariable(popsizeIni, limites, longGen);
    Pop = CromToNro(Pop,limites, longGen);
    popsize = popsizeIni;

    if Dibujo_onLine==1,
        Dibujar(Pop, limites, paso, NroFunc,x,y,z);
    end

    itera=1;
    fitgBest = 0;

    igualMejor = 0;
    Pop = EvaluarFitness(cmax, NroFunc, Pop);

    Pobladores(itera, corrida) = popsize;
    LosMejoresFitness(itera, corrida) = max([Pop.fitness]);
    LosPeoresFitness(itera, corrida) = min([Pop.fitness]);
    LosFitnessPromedio(itera, corrida) = mean([Pop.fitness]);

    [Clases Pop] = AsignarTV(Pop,2);

    while (itera < MAX_ITERA) & (igualMejor < Usar_IgualItera) & (popsize > 0) & (popsize < 300)

        % recordar quien es el mejor y donde esta por el elitismo
        Pop = EvaluarFitness(cmax, NroFunc, Pop);
        [FitMaxAnt, quienAnt] = max([Pop.fitness]);
        MejorIndividuo = Pop(quienAnt);

        Pop = Binary_to_Gray(Pop,limites,longGen);
        rta = generar(Pop, pCross, pMutation, longGen, limites);
        Hijos = rta.newPop;
        cantCross = rta.cantCross;
        cantMut = rta.cantMut;
        Pop = Gray_to_Binary(Pop,limites,longGen);

        Hijos = Gray_to_Binary(Hijos,limites,longGen);
        Hijos = CromToNro(Hijos,limites, longGen);
        Hijos = EvaluarFitness(cmax, NroFunc,Hijos);


        %============ POBLACION VARIABLE ===================
        if (Usar_Poblacion_variable==1),
            %== nacimientos ==
            [CantHijos, sentenciados] = CalcularRadios(Pop);
            %== solos tiene los indices de los que se van a reproducir

            if CantHijos>0,  %== debe haber descendencia ==
                Pop = VerEntorno(Pop, sentenciados, CantHijos);
                [valores,indices] = sort([Hijos.fitness]);
                Hijos = Hijos(indices);

                [fil pos] = size(Hijos);

                for hijo=1:CantHijos,
                    if (pos == 0)
                        [fil pos] = size(Hijos);
                    end
                    Pop = [ Pop Hijos(pos)];
                    pos = pos - 1;
                end

                if itera <= MAX_ITERA_TVProp,
                    [Clases Pop] = AsignarTV(Pop,2);
                else
```

```
            [Clases Pop] = AsignarTV(Pop,1);
        end
    end

    %== Cumpleaños y muertes ==
    debenMorir = [];
    popsize = length(Pop);
    for i=1:popsize
        Pop(i).TV = Pop(i).TV - 1;
        if Pop(i).TV == 0,
            debenMorir = [debenMorir i];
        end
    end

    %== GARANTIZAR QUE LA POBLACION CUMPLE CON EL TAMAÑO MINIMO ==
    if popsize-length(debenMorir) < POPSIZE_MINIMO,
        % == van a quedar muy poquitos, salvemos a los mejores ===
        if popsize<=POPSIZE_MINIMO, %== no puede morir ninguno
            recuperados = debenMorir;
            debenMorir = [];
        else
            puedenMorir = popsize - POPSIZE_MINIMO;
            [aptitudes orden] = sort([Pop(debenMorir).fitness]);
            recuperados = debenMorir(puedenMorir+1:length(debenMorir));
            debenMorir = debenMorir(1:puedenMorir);
        end;
        for i=1:length(recuperados)
            Pop(recuperados(i)).TV = Pop(recuperados(i)).TVIni;
        end
    end

    Pop(debenMorir) = [];
    popsize = length(Pop);
else
    Pop = Hijos;
end    %== de if (Usar_Poblacion_variable==1)

%============== FIN DE POBLACION VARIABLE ====================

%== verificar el elitismo ==
if (elitismo==1),
    [FitnessMax, OrdenDeAptitud] = sort([Pop.fitness]);
    quien = OrdenDeAptitud(popsize);
    FitMax = FitnessMax(popsize);

    %== el mejor (antes de mover a nadie) pasa seguro a la proxima generacion ==
    Pop(OrdenDeAptitud(1)) = MejorIndividuo;

    if (FitMax < FitMaxAnt),
        FitMax = FitMaxAnt;
        quien = OrdenDeAptitud(1);
    end
end

if Dibujo_onLine==1,
    Dibujar(Pop, limites, paso, NroFunc,x,y,z);
end
itera = itera + 1;

[FitMax,quien] = max([Pop.fitness]);
if FitMaxAnt == FitMax
    igualMejor = igualMejor + 1;
else
    igualMejor = 1;
end

Pobladores(itera, corrida) = popsize;
LosMejoresFitness(itera, corrida) = FitMax;
```

```
            LosPeoresFitness(itera, corrida) = min([Pop.fitness]);
            LosFitnessPromedio(itera, corrida) = mean([Pop.fitness]);

        end

        disp(sprintf('Itera = %d, popsize = %d. Las ultimas %d tuvieron fitness = %f',
                        itera,popsize, igualMejor,FitMax));

        estadist(corrida,1) = corrida;
        estadist(corrida,2) = itera;
        estadist(corrida,3) = min([Pop.fitness]);
        estadist(corrida,4) = mean([Pop.fitness]);
        [maximo, quien] = max([Pop.fitness]);
        estadist(corrida,5) = maximo;
        estadist(corrida,6) = popsize;

        estadist(corrida,7:6+longIndiv) = Pop(quien).fenotipo;

    end

    if (Salvar_Datos == 1),
        disp(sprintf('Grabando las %d corridas de la Funcion %d',MAX_CORRIDAS,NroFunc));
        AVGPobladores = avg(Pobladores, 2);
        AVGLosMejoresFitness = avg(LosMejoresFitness, 2);
        AVGLosPeoresFitness = avg(LosPeoresFitness,2);
        AVGLosFitnessPromedio = avg(LosFitnessPromedio,2);

        if ( Usar_Poblacion_variable == 1 ),
            n1 = 'VAR';
            nom2 = sprintf('mediciones\\VAR_Poblacion%d_%d',NroFunc, popsizeIni);
            save(nom2,'AVGPobladores','-ASCII', '-TABS');
        else
            n1 = 'FIX';
        end


        nom1 = sprintf('mediciones\\%s_Fun%d_PopIni_%d', n1,NroFunc,popsizeIni);
        save(nom1,'estadist','-ASCII', '-TABS');

        nom3 = sprintf('mediciones\\%s_Fun%d_PopIni_%d_AVGLosMejores', n1,NroFunc,popsizeIni);
        save(nom3,'AVGLosMejoresFitness','-ASCII', '-TABS');

        nom4 = sprintf('mediciones\\%s_Fun%d_PopIni_%d_AVGLosPeores', n1,NroFunc,popsizeIni);
        save(nom4,'AVGLosPeoresFitness','-ASCII', '-TABS');

        nom5 = sprintf('mediciones\\%s_Fun%d_PopIni_%d_AVGLosMedios', n1,NroFunc,popsizeIni);
        save(nom5,'AVGLosFitnessPromedio','-ASCII', '-TABS');

        %== solo para verificar ==
        nom6 = sprintf('mediciones\\detalles\\%s_Fun%d_PopIni_%d_LosMejoresFitness',
                        n1,NroFunc,popsizeIni);
        save(nom6,'LosMejoresFitness','-ASCII', '-TABS');

        nom7 = sprintf('mediciones\\detalles\\%s_Fun%d_PopIni_%d_LosPeoresFitness',
                        n1,NroFunc,popsizeIni);
        save(nom7,'LosPeoresFitness','-ASCII', '-TABS');

        nom8 = sprintf('mediciones\\detalles\\%s_Fun%d_PopIni_%d_LosFitnessPromedio',
                        n1,NroFunc,popsizeIni);
        save(nom8,'LosFitnessPromedio','-ASCII', '-TABS');
    end
end



function [cmax, limites, paso, valorElegido, x,y,z] = FuncionElegida(nro);
```

```
valorElegido = nro;
x = [];
y = [];
z = [];
switch valorElegido
    case 1
        %===== f(x,y) = x^2 + y^2  ======
        cmax = 50; %Valor maximo utilizado para calcular el fitness
        limites = [-1 -1; 5 5]; % c/columna representa el rango en el que varia cada argumento de la funcion
        paso = 0.1;   %Precision para el dibujo de la funcion
        [x,y] = meshgrid(limites(1,1):paso:limites(2,1), limites(1,2):paso:limites(2,2));
        z= x.^2 + y.^2;

    case 2
        %===== f(x) = sin(x * pi ...) ======
        cmax = 3; %Valor maximo utilizado para calcular el fitness
        limites = [-2; 1]; % c/columna representa el rango en el que varia cada argumento de la funcion
        paso = 0.005;   %Precision para el dibujo de la funcion
        z = 0.5 + (sin(sqrt(x.^2 + y.^2)) .^ 2 - 0.5) ./ ((1 + 0.001 .* (x.^2 + y.^2)).^2);

    case 3
        %===== f(x,y) = 0.5+((sin(sqrt(x^2+y^2))^2-0.5)/(1+0.001*(x^2+y^2))^2) ======
        cmax = 1; %Valor maximo utilizado para calcular el fitness
        limites = [-50 -50; 50 50]; % c/columna representa el rango en el que varia cada argumento de
                                    % la funcion
        paso = 0.25;   %Precision para el dibujo de la funcion
        [x,y] = meshgrid(limites(1,1):paso:limites(2,1), limites(1,2):paso:limites(2,2));
        z = 0.5+((sin( sqrt(x.^2 + y.^2 + 4) ).^2-0.5)./(1+0.001*(x.^2 + y.^2)).^2);

    case 4
        %===== f(x,y) = 1/0.002 + sumatoria (50*j + (x(i,j)-a(1,k))^6 + (y(i,j)-a(2,k))^6) ======
        cmax = 500; %Valor maximo utilizado para calcular el fitness
        limites = [-50 -50; 50 50]; % c/columna representa el rango en el que varia cada argumento
                                    % de la funcion
        paso = 1;   %Precision para el dibujo de la funcion
        [x,y] = meshgrid(limites(1,1):paso:limites(2,1), limites(1,2):paso:limites(2,2));
        a = [ -32 -16 0 16 32 -32 -16 0 16 32 -32 -16 0 16 32 -32 -16 0 16 32 -32 -16 0 16 32
              -32 -32 -32 -32 -32 -16 -16 -16 -16 -16 0 0 0 0 0 16 16 16 16 16 32 32 32 32 32];
        [fil,col] = size(x) ;
        for i=1:fil
            for j=1:col
                suma = 0;
                for k=1:25
                    suma = suma + 1/(50*k + (x(i,j)-a(1,k))^6 + (y(i,j)-a(2,k))^6);
                end
                z(i,j)=1/(0.002+suma);
            end;
        end;

    case 5
        %===== f(x,y) = 1/(0.002+(1./(1+(x-1)^12+(y+1)^12))) ======
        cmax = 500; %Valor maximo utilizado para calcular el fitness
        limites = [-200 -200; 200 200]; % c/columna representa el rango en el que varia cada argumento
                                        % de la funcion
        paso = 2;   %Precision para el dibujo de la funcion
        [x,y] = meshgrid(limites(1,1):paso:limites(2,1), limites(1,2):paso:limites(2,2));
        z = 1./(0.002+(1./(1+(x-1).^12+(y+1).^12)));

    case 6
        %===== f(x,y) = 1/0.002 + sumatoria ((50*j)^2 + (x(i,j)-a(1,k))^12 + (y(i,j)-a(2,k))^12) ======
        cmax = 500; %Valor maximo utilizado para calcular el fitness
        limites = [-50 -50; 50 50]; % c/columna representa el rango en el que varia cada argumento
                                    % de la funcion
        paso = 1;   %Precision para el dibujo de la funcion
        [x,y] = meshgrid(limites(1,1):paso:limites(2,1), limites(1,2):paso:limites(2,2));
        a = [ -30  16  30; -40 -32 35];
        [fil,col] = size(x) ;
        for i=1:fil
```

```
            for j=1:col
                suma = 0;
                for k=1:3
                    suma = suma + 1/(50*k^2 + (x(i,j)-a(1,k))^12 + (y(i,j)-a(2,k))^12);
                end
                z(i,j)=1/(0.002+suma);
            end;
        end;

end
```

```
function p = CrearPoblacionInicialVariable(popsize, limites, longGen)
% genera un vector de popsize inidividuos aleatorios
% cada uno es una estructura con los valores necesarios

long = size(limites,2); %cantidad de argumentos de la funcion
p = [];
nuevo = zeros(longGen,long);
fenotipo = zeros(1,long);

for i=1:popsize
    nuevo = round(rand(longGen,long));
    p = [p struct('genotipo', nuevo, 'fenotipo', fenotipo, 'fitness', 0 , 'TV', 0 , 'TVIni',0)];
end
```

```
function y = CromToNro(x, limites, longGen)

% funcion que convierte los puntos en representacion binaria a su representacion real
% x es la poblacion
% y es una matriz con el valor de los fenotipos

y = x;
valorMax = 2^longGen - 1;
long = size(limites,2); %cantidad de argumentos de la funcion
[fila tam] = size(x);

for i=1:1:tam
    for j = 1:1:long,
        % armo la primer coordenada
        nro=0;
        %el primer digito representa la potencia mas pequenia y el ultimo la mas alta
        for k=1:1:longGen
            nro = nro + (2^(k-1))*y(i).genotipo(k,j);
        end
        % paso la primer coordenada a su representacion real
        y(i).fenotipo(j) = limites(1,j) + (nro/valorMax) * (limites(2,j) - limites(1,j));
    end
end
```

```
function nada = Dibujar(Pop, limites, paso, NroFunc, x, y, z);

CantArgs = size(limites,2);

switch CantArgs
    case 1  % dibujo 2D
        hold off
        %Visualizar la funcion FUN en el intervalo indicado
        Intervalo = limites(1,1):paso:limites(2,1);
```

```
        y = FUN(NroFunc, Intervalo);
        plot(Intervalo,y);
        hold on
        % graficar los individuos

        Salida = FUN(NroFunc, [Pop.fenotipo]);
        plot([Pop.fenotipo],Salida,'*');
        pause(0.01);
    case 2 %dibujo 3D
        hold off
        mesh(x,y,z);

        hold on
        z=FUN(NroFunc, [Pop.fenotipo]);
        puntos = [Pop.fenotipo];
        long = length(puntos);
        puntosx = puntos([1:2:long]);
        puntosy = puntos([2:2:long]);

        plot3(puntosx, puntosy, z, '*');
        pause(0.01);
end
```

```
function Pop = EvaluarFitness(cmax, NroFunc, Pop)

% evaluar el fitness de todos los individuos
popsize = length(Pop);
Salida = FUN(NroFunc, [Pop.fenotipo]);
Fitness = abs(cmax * ones(1,popsize) - Salida);

for i=1:popsize
    Pop(i).fitness = Fitness(i);
end
```

```
function y = FUN(nro, x)
% x es una matriz con tantas columnas como puntos donde debe evaluarse la
% funcion y tantas filas como variables de entrada tenga la funcion
y = [];
long = length(x);
switch nro
    case 1
        puntosx = x([1:2:long]);
        puntosy = x([2:2:long]);
        y = puntosx.^2 + puntosy.^2;

    case 2
        y = ((-x) .* sin(10 * pi * x) )+ ones(1,length(x));

    case 3
        puntosx = x([1:2:long]);
        puntosy = x([2:2:long]);
        y = 0.5+((sin( sqrt(puntosx.^2+puntosy.^2 + 4) ).^2-0.5)./(1+0.001*(puntosx.^2+puntosy.^2)).^2);

    case 4
        puntosx = x([1:2:long]);
        puntosy = x([2:2:long]);
        a = [ -32 -16 0 16 32 -32 -16 0 16 32 -32 -16 0 16 32 -32 -16 0 16 32 -32 -16 0 16 32
            -32 -32 -32 -32 -32 -16 -16 -16 -16 -16 0 0 0 0 0 16 16 16 16 16 32 32 32 32 32];
        long = length(puntosx);
        for i=1:long
            suma = 0;
            for k=1:25
```

```
                suma = suma + 1/(50*k+(puntosx(i)-a(1,k))^6+(puntosy(i)-a(2,k))^6);
            end
            y(i)=1/(0.002+suma);
        end;

    case 5
        puntosx = x([1:2:long]);
        puntosy = x([2:2:long]);
        y = 1./(0.002+(1./(1+(puntosx-1).^12+(puntosy+1).^12)));

    case 6
        puntosx = x([1:2:long]);
        puntosy = x([2:2:long]);
        a = [ -30  16  30; -40 -32 35];
        long = length(puntosx);
        for i=1:long
            suma = 0;
            for k=1:3
                suma = suma + 1/(50*k^2+(puntosx(i)-a(1,k))^12+(puntosy(i)-a(2,k))^12);
            end
            y(i)=1/(0.002+suma);
        end;

end
```

```
function [Centros, Pop] = AsignarTV(Pop, tipo);
% TIPO vale 1 si se usa asignacion de TV fijo por clase y 2 si se usa
% asignacion de TV proporcional a la cant.de individuos por clase

CANT_CLASES = 4;
MAX_TV = 10;

popsize = length(Pop);

%=== Asignar los centros de las clases iniciales ===
% los primeros centros son random
mezcla= randperm(popsize);

Centros = zeros(1,CANT_CLASES);
CANT = min(CANT_CLASES, popsize);
for i= 1:CANT
    Centros(1,i) = Pop(mezcla(i)).fitness;
end


actual = zeros(1,popsize);
%== recalcular los centros hasta que se estabilicen
dife = 10;
while dife>1
    ant = actual;

    Sumas = zeros(1, CANT_CLASES);
    Cant = zeros(1,CANT_CLASES);
    for i=1:popsize
        clase = ClaseMasCercana(Pop(i).fitness, Centros);
        Sumas(1,clase) = Sumas(1,clase) + Pop(i).fitness;
        Cant(1,clase) = Cant(1,clase) + 1;
        actual(i) = clase;
    end

    %== recalcular los centros ==
    for k = 1:CANT_CLASES
        if Cant(k)>0,
            Centros(1,k) = Sumas(1,k) / Cant(k);
```

```
        end
    end
    dife = sum(abs(ant-actual));
end

%== ordenar los centroides en forma creciente por el fitness ==
[Centros orden] = sort(Centros);
Cant = Cant(orden);
actual = orden(actual);

%== 'Centros' contiene los centroides y 'Cant' la cant.de elementos en c/clase ==
%== 'actual' indica la clase de cada individuo ==
%== Calcular el fitness minimo y maximo por clase ==
FitMin = zeros(CANT_CLASES);
FitMax = zeros(CANT_CLASES);
for k = 1:CANT
    elems=[];
    elems = find(actual == k);
    if length(elems) > 0,
        FitMin(k) = min([Pop(elems).fitness]);
        FitMax(k) = max([Pop(elems).fitness]);
    end
end

% FALTA ASIGNAR LOS TV segun lo que diga TIPO.
cuales = [];
cuales = find([Pop.TV] == 0);

if tipo==1
    %==  TV fijo por clase ==
    AnchoClase = MAX_TV / CANT_CLASES;
    for i=1:length(cuales)
        c = actual(cuales(i));
        TVClasesAnt = (c - 1)*AnchoClase;

        if FitMax(c)== FitMin(c),
            despl = 1;
        else
            despl = (Pop(cuales(i)).fitness - FitMin( c ))/abs(FitMax(c)-FitMin(c));
        end

        Pop(cuales(i)).TV = max(1, floor(TVClasesAnt + AnchoClase * despl ));
        Pop(cuales(i)).TVIni = Pop(cuales(i)).TV;
    end
else
    %== TV proporcional a la cant.de individuos ==
    for i=1:length(cuales)
        c = actual(cuales(i));
        elemsAnt = sum(Cant(1:c-1));
        TVAnterior = MAX_TV * elemsAnt / popsize;
        TVClaseActual = MAX_TV * Cant(c) / popsize;
        if FitMax(c)== FitMin(c),
            despl = 1;
        else
            despl = (Pop(cuales(i)).fitness - FitMin( c )) / abs(FitMax(c)-FitMin(c));
        end

        Pop(cuales(i)).TV = max(1,floor(TVAnterior + TVClaseActual * despl ));
        Pop(cuales(i)).TVIni = Pop(cuales(i)).TV;
    end
end



function masCerca = ClaseMasCercana(quien, Centros);
%== calcula distancia euclidea entre 'quien' y c/u de las columnas de 'Centros'
[fil,col] = size(Centros);
```

```
distancias = sqrt(sum((Centros - quien*ones(1,col)).^2,1));
[menor,masCerca] = min(distancias);



function Pop = VerEntorno(Pop, sentenciados, CantNuevos);
%== sentenciados es un vector fila con los indices de los individuos muy proximos ==
%== tambien puede ser que sean el vecino del vecino del vecino ==
%== solo debe permanecer en la poblacion el mejor ==
sobrevivientes = 2;
if length(sentenciados) > sobrevivientes,
    CantBorrados = min( length(sentenciados)-sobrevivientes, CantNuevos);

    [ValoresFit, orden] = sort([Pop(sentenciados).fitness]);

    sentenciados(orden(CantBorrados+1: length(orden))) = [];
    %== solo quedan los que van a sobrevivir ==

    Pop(sentenciados) = [];
end



function [CantHijos, sentenciados]= CalcularRadios(Pop)
% calcular la matriz de distancias entre los individuos de la poblacion

popsize = length(Pop);
if popsize < 3,
    CantHijos = 3;
    sentenciados = [];
else
    %== retirar del analisis el 10% de los mejores individuos ==

    d=Distancias(Pop); %= matriz de distancias ==
    masLejos = max(max(d));
    for i = 1 : popsize
        d(i,i)=masLejos;
    end
    [radiosMinimos, menores] = min(d);

    media = mean(radiosMinimos);
    desviacion = sum(abs(radiosMinimos - media))/popsize;

    CantHijos = length(find(radiosMinimos > media));

    sentenciados = find((radiosMinimos < (media-desviacion)));
end



function d = Distancias(Pop)

popsize = length(Pop);

if popsize < 2,
    d = 0;
else
    d = zeros(popsize,popsize);
    P=[];
    for i = 1 : popsize
        P = [P [Pop(i).fenotipo]'];
    end

    for i = 1 : popsize-1
        p1 = P(:,1);
```

```
        P(:,1)=[];
        [fil,col] = size(P);

        distancias = sqrt(sum((P - p1*ones(1,col)).^2,1));
        d(i,i+1:popsize) = distancias;
        d(i+1:popsize, i) = distancias';
    end
end



function y = Binary_to_Gray(x,limites,longGen)

% Esta funcion convierte los puntos en representacion binaria a codigos gray para el cruce y mutacion

poblSize=size(x);
long = size(limites,2); %cantidad de argumentos de la funcion
y=x;
for i=1:1:poblSize
    for j=1:1:long
        for k=1:1:longGen - 1
            y(i).genotipo(k,j) = xor(y(i).genotipo(k,j), y(i).genotipo(k + 1,j));
        end
    end
end



function rta = generar(poblacion, pCross, pMutation,longGen,limites)

[fil poblSize] = size(poblacion);
long = size(limites,2); %cantidad de argumentos de la funcion
y = [];
cantCrossOver=0;
cantMutaciones=0;

% por c/iteracion se generan dos hijos
for i=1:2:poblSize,
    p1 = select([poblacion.fitness]);           % seleccionar la posición de
    p2 = select([poblacion.fitness]);           % los dos padres
    % generar los dos hijos
    rta2 = cruzar_y_mutar(poblacion,p1,p2,pCross,pMutation,longGen,limites);
    h1 = rta2.hijo1;
    h2 = rta2.hijo2;
    cantCrossOver = cantCrossOver + rta2.cantCross;
    cantMutaciones = cantMutaciones + rta2.cantMut;

    y = [y h1 h2];

end
rta = struct('newPop',y,'cantCross',cantCrossOver,'cantMut',cantMutaciones);



function rta2 = cruzar_y_mutar(poblacion,p1,p2,pCross,pMutation,longGen,limites)

poblSize = size(poblacion);
long = size(limites,2); %cantidad de argumentos de la funcion
h1=poblacion(p1);
h2=poblacion(p2);
cantCrossOver=0;
cantMutaciones=0;
%ver si corresponde aplicar crossover
hayCrossover = ((pCross==1) | (rand<=pCross));
if hayCrossover,
```

```
    posicion = round(rand * (round(longGen/2)-2)) + 1;
    cantCrossOver=cantCrossOver+1;;
else
    posicion = longGen;
end
for i=1:1:long
    for j=1:1:posicion,
        rta3 = mutar(poblacion(p1).genotipo(j,i),pMutation);
        h1.genotipo(j,i) = rta3.gen;
        cantMutaciones=cantMutaciones+rta3.mut;
        rta3 = mutar(poblacion(p2).genotipo(j,i),pMutation);
        h2.genotipo(j,i) = rta3.gen;
        cantMutaciones=cantMutaciones+rta3.mut;
    end
end

if hayCrossover,
    for i=1:1:long
        for j=posicion+1:1:longGen,
            rta3 = mutar(poblacion(p2).genotipo(j,i),pMutation);
            h1.genotipo(j,i) = rta3.gen;
            cantMutaciones=cantMutaciones+rta3.mut;
            rta3 = mutar(poblacion(p1).genotipo(j,i),pMutation);
            h2.genotipo(j,i) = rta3.gen;
            cantMutaciones=cantMutaciones+rta3.mut;
        end
    end
end

rta2 = struct('hijo1',h1,'hijo2',h2,'cantCross',cantCrossOver,'cantMut',cantMutaciones);



function y = select(fitnessPob)

% funcion que evalua el fitness de cada individuo y selecciona uno por el metodo de la ruleta

cuantos = length(fitnessPob);
sumFitness = sum(fitnessPob);
aleatorio = rand * sumFitness;  % posicion dentro de la ruleta
suma = fitnessPob(1);
j = 1;
while (suma > aleatorio) & (j<cuantos),
    j = j + 1;
    suma = suma + (fitnessPob(j));
end

y = j;                  % posición   elegida



function rta3 = mutar(d1,pMutacion)

% funcion que realiza la mutacion del bit pasado como parametro con probabilidad de mutacion pMutacion

hayMutacion =  (rand<=pMutacion);
mut=0;
if hayMutacion,
    mut=1;
    if (d1==0)
        d=1;
    else d=0;
    end
else d=d1;
end
rta3 = struct('gen',d,'mut',mut);
```

```
function y=Gray_to_Binary(x,limites,longGen)

% Esta funcion convierte los puntos en representacion de codigos gray a binaria
% para evaluar los fenotipos


poblSize=size(x);
long = size(limites,2); %cantidad de argumentos de la funcion
y=x;
for i=1:1:poblSize
    for j=1:1:long
        for k=longGen:-1:2
            y(i).genotipo(k-1,j) = xor(y(i).genotipo(k-1,j), y(i).genotipo(k,j));
        end
    end
end




function mean = avg(x, dim)

mean = sum(x .* (x>-1),2)./ (sum(x>-1,2) + (sum(x>-1,2) == 0));
```

Índice de figuras

Índice de cuadros

Bibliografía

[1] E. Alba. *Parallel Metaheuristics: A New Class of Algorithms.* John Wiley and Sons, 2005.

[2] D. Maravall-Gómez Allende. *Reconocimiento de formas y Visión artificial.* Addison Wesley, 1994.

[3] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. *In Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 14—-22, 1987.

[4] T. Bäck, D. Fogeln, and Z. Michalewicz. *Handbook of Evolutionary Computation.* IOP Publishing and Oxford University Press, 1997.

[5] F. Van Den Bergh. *An analysis of particle swarm optimizers.* PhD thesis, Department Computer Science. University Pretoria. South Africa, 2002.

[6] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[7] L. Booker. *Intelligent Behavior as an Adaptation to the Task Environment.* PhD thesis, University of Michigan, 1982.

[8] A. Brindle. *Genetic Algorithms for Function Optimization.* PhD thesis, University of Alberta, 1981.

[9] M. Clerc. *Binary Particle Swarm Optimisers: Toolbox, Derivations, and Mathematical Insights.* 2005.

[10] M. Clerc and J. Kennedy. The particle swarm – explosion, stability and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6:58–73, 2002.

[11] C. Coello Coello. *Introducción a la Computación Evolutiva.* Abril 2006.

[12] C. Coello Coello, A. Christiansen, and A. Hernández Aguirre. Using a new ga-based multiobjective optimization technique for the design of robot arms. *Robotica*, 16(4):401–414, 1998.

[13] C. Coello Coello, F. Santos Hernández, and F. Alonso Farrera. Using a new ga-based multiobjective optimization technique for the design of robot arms. *Robotica*, 16(4):401–414, 1998.

[14] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[15] M Dorigo. *Optimization, Learning and Natural Algorithms.* PhD thesis, Dipartamento di Elettronica, Politecnico di Milano, 1992.

[16] R. Eberhart and Y. Shi. Comparing inertia weights and constriction factors in particle swarm optimization. *In Proceedings of the International Congress on Evolutionary Computation*, 1:84–88, 2000.

[17] L. Ertöz, M. Steinbach, and V. Kumar. A new shared nearest neighbor clustering algorithm and its applications. *Proc. Workshop on Clustering High Dimensional Data and its Applications*, pages 105–115, 2002.

[18] T. Feo and M. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optmization*, 6:109–133, 1999.

[19] C. Fernandes, V. Ramos, and A. Rosa. Varying the population size of artificial foraging swarm on time varying landscapes. *Artificial Neural Networks: Biological Inspirations, Proc. ICANN´05: 15th Int. Conf.*, 3696:311–316, 2005.

[20] A. Fiszelew. *Generación Automática de Redes Neuronales con Ajuste de Parámetros basado en Algoritmos Genéticos.* PhD thesis, Universidad De Buenos Aires, 2002.

[21] F. Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8:156–166, 1977.

[22] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13:533–549, 1986.

[23] F. Glover and G. Kochenberger. *Handbook of Metaheuristics*. Kluwer Academic Publishers, 2002.

[24] Goldberg. *Foundations of Genetic Algorithms*, chapter A comparative analysis of selection schemes used in genetic algorithms, pages 69–93. Morgan Kaufmann, 1991.

[25] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing, 1989.

[26] A. Hoffman. *Arguments on Evolution: A Paleontologist's Perspective*. Oxford University Press, New York, 1989.

[27] J. Holland. *Concerning efficient adaptive systems*. Spartan Books, 1962.

[28] J. Holland. Outline for a logical theory of adaptive systems. *Journal of the Association for Computing Machinery*, 9:297—-314, 1962.

[29] J. Holland. *Adaptation in Natural and Artificial Systems*. The MIT Press, 1975.

[30] Jain. *Handbook of Pattern Recognition and Image Processing*, chapter Cluster Analysis, pages 33–57. Academic Press, 1986.

[31] A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975.

[32] L. Kaufman and P.J. Rousseuw. *Finding Groups in data*. John Wiley, 1990.

[33] J. Kennedy and R. Eberhart. Particle swarm optimization. *In Proceedings of the IEEE International Conference on Neural Networks*, 4:1942–1948, 1995.

[34] J. Kennedy and R. Eberhart. A discrete binary version of the particle swarm algorithm. *In Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, 5:4104–4109, 1997.

[35] J. Kennedy, R. Eberhart, and Y. Shi. *Swarm Intelligence*. Morgan Kaufmann Publishers, 2001.

[36] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[37] J. Larry, Eshelman, and J. Davis Schaffer. *Real-coded Genetic Algorithms and Interval-Schemata*. Van Nostrand Reinhold, 1991.

[38] L.Lanzarini, C. Sanz, M.Ñaiouf, and M. Romero. Mixed alternative in the assignment by classes vs conventional methods for calculation of individuals lifetime in gavaps. *Proceedings of the 22nd International Conference on Information Technology Interfaces, ITI 2000*, 953-96769-1-6:383–389, 2000.

[39] R. Marti. Procedimientos metaheurísticos en optimización combinatoria.

[40] García Martínez, Servente, and Pasquini. *Sistemas Inteligentes*. Nueva Librería, 2003.

[41] M. Meissner, M. Schmuker, and G. Schneider. Optimized particle swarm optimization (opso) and its application to artificial neural networks training. *BMC Bioinformatics 2006*, pages 7–125, 2006.

[42] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springers-Verlag, 1994.

[43] N. Mladenovic and P. Hansen. Variable neighborhood search. *Computers Operation*, 24:1097–1100, 1997.

[44] A. Kuri Morales. Algoritmos genéticos. Ciencia de la computación, 2002.

[45] J. García Nieto. *Algoritmos basados en Cúmulos de Partículas para la resolución de problemas complejos*. PhD thesis, Universidad de Málaga, 2006.

[46] IUPAC-IUB Commission on BiochemicalÑomenclature. Abbreviations and symbols for nucleic acids, polynucleotides and their constituents, 1974.

[47] C. B. Reeves. *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley and Sons, 1993.

[48] G. Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Transactions on Neural Networks*, 5:96–101, 1994.

[49] S.Bird and X. Li. Adaptively choosing niching parameters in a pso. *Proceeding of Genetic and Evolutionary Computation Conference 2006 (GECCO'06), eds. M Keijzer*, ACM Press:3–9, 2006.

[50] J. D. Schaffer and A. Morishima. An adaptative crossover distribution mecanism for genetic algorithms. *Proceedings of the Second International Conference on Genetic Algorithms*, pages 36–40, 1987.

[51] Y. Shi and R. Eberhart. Parameter selection in particle swarm optimization. *Proceedings of the 7th International Conference on Evolutionary Programming*, pages 591–600, 1998.

[52] T. Stützle. *Local Search Algorithms for Combinatorial Problems Analysis, Algorithms and New Applications*. Technical report, DISKI Dissertationen zur Künstliken Intelligenz, 1999.

[53] J. Watson and F. Crick. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 1953.

[54] A. Wetzel. *Evaluation og the Effectiveness of Genetics Algorithms in Combinatorial Optimization*. PhD thesis, University of Pittsburgh, 1983.

[55] D. Whitley. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. *In Proceedings of the Third International Conference on Genetic Algorithms*, pages 116—-121, 1989.

[56] A. Wright. *Foundations of Genetic Algorithms*, chapter Genetic Algorithms for Real Parameter Optimization, pages 205–218. Morgan Kaufmann Publishers, 1991.